New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice

CRISTIAN ESTAN and GEORGE VARGHESE
University of California, San Diego

Accurate network traffic measurement is required for accounting, bandwidth provisioning and detecting DoS attacks. These applications see the traffic as a collection of flows they need to measure. As link speeds and the number of flows increase, keeping a counter for each flow is too expensive (using SRAM) or slow (using DRAM). The current state-of-the-art methods (Cisco's sampled NetFlow) which count periodically sampled packets are slow, inaccurate and resource-intensive. Previous work showed that at different granularities a small number of "heavy hitters" accounts for a large share of traffic. Our paper introduces a paradigm shift by concentrating the measurement process on large flows only — those above some threshold such as 0.1% of the link capacity.

We propose two novel and scalable algorithms for identifying the large flows: *sample and hold* and *multistage filters*, which take a constant number of memory references per packet and use a small amount of memory. If M is the available memory, we show analytically that the errors of our new algorithms are proportional to $1/M$; by contrast, the error of an algorithm based on classical sampling is proportional to $1/\sqrt{M}$, thus providing much less accuracy for the same amount of memory. We also describe further optimizations such as *early removal* and *conservative update* that further improve the accuracy of our algorithms, as measured on real traffic traces, by an order of magnitude. Our schemes allow a new form of accounting called *threshold accounting* in which only flows above a threshold are charged by usage while the rest are charged a fixed fee. Threshold accounting generalizes usage-based and duration based pricing.

Categories and Subject Descriptors: C.2.3 [**Computer-Communication Networks**]: Network Operations—*traffic measurement, identifying large flows*

General Terms: Algorithms,Measurement

Additional Key Words and Phrases: Network traffic measurement, usage based accounting, scalability, on-line algorithms, identifying large flows

1. INTRODUCTION

> *If we're keeping per-flow state, we have a scaling problem, and we'll be tracking millions of ants to track a few elephants.* — Van Jacobson, End-to-end Research meeting, June 2000.

Measuring and monitoring network traffic is required to manage today's complex Internet backbones [Feldmann et al. 2000; Duffield and Grossglauser 2000]. Such measurement information is essential for short-term monitoring (e.g., detecting hot spots and denial-of-service attacks [Mahajan et al. 2001]), longer term traffic engineering (e.g., rerouting traffic[Shaikh et al. 1999] and upgrading selected links[Feldmann et al. 2000]), and accounting (e.g., to support usage based pric-

Author emails: cestan@cs.ucsd.edu and varghese@cs.ucsd.edu

ing[Duffield et al. 2001]).

The standard approach advocated by the Real-Time Flow Measurement (RTFM) [Brownlee et al. 1999] Working Group of the IETF is to instrument routers to add flow meters at either all or selected input links. Today's routers offer tools such as NetFlow [NetFlow] that give flow level information about traffic.

The main problem with the flow measurement approach is its lack of *scalability*. Measurements on MCI traces as early as 1997 [Thomson et al. 1997] showed over 250,000 concurrent flows. More recent measurements in [Fang and Peterson 1999] using a variety of traces show the number of flows between end host pairs in a one hour period to be as high as 1.7 million (Fix-West) and 0.8 million (MCI). Even with aggregation, the number of flows in 1 hour in the Fix-West used by [Fang and Peterson 1999] was as large as 0.5 million.

It can be feasible for flow measurement devices to keep up with the increases in the number of flows (with or without aggregation) only if they use the cheapest memories: DRAMs. Updating per-packet counters in DRAM is already impossible with today's line speeds; further, the gap between DRAM speeds (improving 7-9% per year) and link speeds (improving 100% per year) is only increasing. Cisco NetFlow [NetFlow], which keeps its flow counters in DRAM, solves this problem by sampling: only sampled packets result in updates. But Sampled NetFlow has problems of its own (as we show later) since sampling affects measurement accuracy.

Despite the large number of flows, a common observation found in many measurement studies (e.g., [Feldmann et al. 2000; Fang and Peterson 1999]) is that a small percentage of flows accounts for a large percentage of the traffic. [Fang and Peterson 1999] shows that 9% of the flows between AS pairs account for 90% of the byte traffic between all AS pairs.

For many applications, knowledge of these large flows is probably sufficient. [Fang and Peterson 1999; Pan et al. 2001] suggest achieving scalable differentiated services by providing selective treatment only to a small number of large flows. [Feldmann et al. 2000] underlines the importance of knowledge of "heavy hitters" for decisions about network upgrades and peering. [Duffield et al. 2001] proposes a usage sensitive billing scheme that relies on exact knowledge of the traffic of large flows but only samples of the traffic of small flows.

We conclude that it is infeasible to accurately measure all flows on high speed links, but many applications can benefit from accurately measuring only the few large flows. One can easily keep counters for a few large flows using a small amount of fast memory (SRAM). However, how does the device know which flows to track? If one keeps state for *all* flows to identify the *few* large flows, our purpose is defeated.

Thus a reasonable goal is to devise an algorithm that identifies large flows *using memory that is only a small constant larger than is needed to describe the large flows in the first place*. This is the central question addressed by this paper. We present two algorithms that provably identify large flows using such a small amount of state. Further, our algorithms use only a few memory references per packet, making them suitable for use in high speed routers. Our algorithms produce more accurate estimates than Sampled NetFlow, but they do processing and access memory for each packet. Therefore the small amount of memory they use has to be fast memory operating at line speeds.

1.1 Problem definition

A flow is generically defined by an optional *pattern* (which defines which packets we will focus on) and an *identifier* (values for a set of specified header fields). We can also generalize by allowing the identifier to be a *function* of the header field values (e.g., using prefixes instead of addresses based on a mapping using route tables). Flow definitions vary with applications: for example for a traffic matrix one could use a wildcard pattern and identifiers defined by distinct source and destination network numbers. On the other hand, for identifying TCP denial of service attacks one could use a pattern that focuses on TCP packets and use the destination IP address as a flow identifier. Note that we do not require the algorithms to support simultaneously all these ways of aggregating packets into flows. The algorithms know a priori which flow definition to use and they do not need to ensure that a posteriori analyses based on different flow definitions are possible (as they are based on NetFlow data).

Large flows are defined as those that send more than a given threshold (say 0.1% of the link capacity) during a given measurement interval (1 second, 1 minute or even 1 hour). The technical report version of this paper [Estan and Varghese 2002] gives alternative definitions and algorithms based on defining large flows via leaky bucket descriptors.

An ideal algorithm reports, at the end of the measurement interval, the flow IDs and sizes of all flows that exceeded the threshold. A less ideal algorithm can fail in three ways: it can omit some large flows, it can wrongly add some small flows to the report, and can give an inaccurate estimate of the traffic of some large flows. We call the large flows that evade detection *false negatives*, and the small flows that are wrongly included *false positives*.

The minimum amount of memory required by an ideal algorithm is the inverse of the threshold; for example, there can be at most 1000 flows that use more than 0.1% of the link. We will measure the performance of an algorithm by four metrics: first, its memory compared to that of an ideal algorithm; second, the algorithm's probability of false negatives; third, the algorithm's probability of false positives; and fourth, the expected error in traffic estimates.

1.2 Motivation

Our algorithms for identifying large flows can potentially be used to solve many problems. Since different applications define flows by different header fields, we need a separate instance of our algorithms for each of them. Applications we envisage include:

—**Scalable Threshold Accounting:** The two poles of pricing for network traffic are usage based (e.g., a price per byte for each flow) or duration based (e.g., a fixed price based on duration). While usage-based pricing [Mackie-Masson and Varian 1995; Shenker et al. 1996] has been shown to improve overall utility, usage based pricing in its most complete form is not scalable because we cannot track all flows at high speeds. We suggest, instead, a scheme where we measure all aggregates that are above z% of the link; such traffic is subject to usage based pricing, while the remaining traffic is subject to duration based pricing. By varying z from 0 to 100, we can move from usage based pricing to duration based pricing. More importantly, for reasonably small values of z (say 1%) threshold accounting may offer a compromise between that is scalable and yet offers almost the same utility as usage based pricing. [Altman and Chu 2001] offers experimental evidence based on the INDEX experiment that such threshold pricing could be attractive to both users and ISPs. [1]

—**Real-time Traffic Monitoring:** Many ISPs monitor backbones for hot-spots in order to identify large traffic aggregates that can be rerouted (using MPLS tunnels or routes through optical switches) to reduce congestion. Also, ISPs may consider sudden increases in the traffic sent to certain destinations (the victims) to indicate an ongoing attack. [Mahajan et al. 2001] proposes a mechanism that reacts as soon as attacks are detected, but does not give a mechanism to detect ongoing attacks. For both traffic monitoring and attack detection, it may suffice to focus on large flows.

—**Scalable Queue Management:** At a smaller time scale, scheduling mechanisms seeking to approximate max-min fairness need to detect and penalize flows sending above their fair rate. Keeping per flow state only for these flows [Feng et al. 2001; Pan et al. 2001] can improve fairness with small memory. We do not address this application further, except to note that our techniques may be useful for such problems. For example, [Pan et al. 2001] uses classical sampling techniques to estimate the sending rates of large flows. Given that our algorithms have better accuracy than classical sampling, it may be possible to provide increased fairness for the same amount of memory by applying our algorithms.

The rest of the paper is organized as follows. We describe related work in Section 2, describe our main ideas in Section 3, and provide a theoretical analysis in Section 4. We theoretically compare our algorithms with NetFlow in Section 5. After showing how to dimension our algorithms in Section 6, we describe experimental evaluation on traces in Section 7. We end with implementation issues in Section 8 and conclusions in Section 9.

2. RELATED WORK

The primary tool used for flow level measurement by IP backbone operators is Cisco NetFlow [NetFlow]. NetFlow keeps per flow counters in a large, slow DRAM. Basic NetFlow has two problems: **i) Processing Overhead:** updating the DRAM slows down the forwarding rate; **ii) Collection Overhead:** the amount of data generated by NetFlow can overwhelm the collection server or its network connection. For example [Feldmann et al. 2000] reports loss rates of up to 90% using basic NetFlow.

The processing overhead can be alleviated using sampling: per-flow counters are incremented *only* for sampled packets[2]. Classical random sampling introduces considerable inaccuracy in the estimate; this is not a problem for measurements over long periods (errors average out) and if applications do not need exact data.

[1]Besides [Altman and Chu 2001], a brief reference to a similar idea can be found in [Shenker et al. 1996]. However, neither paper proposes a fast mechanism to implement the idea.

[2]NetFlow preforms 1 in N periodic sampling, but to simplify the analysis we assume in this paper that it performs ordinary sampling processing each packet with probability 1/N independently.

However, we will show that sampling does not work well for applications that require true lower bounds on customer traffic (e.g., it may be infeasible to charge customers based on estimates that are *larger* than actual usage) and for applications that require accurate data at small time scales (e.g., billing systems that charge higher during congested periods).

The data collection overhead can be alleviated by having the router aggregate flows (e.g., by source and destination AS numbers) as directed by a manager. However, [Fang and Peterson 1999] shows that even the number of aggregated flows is very large. For example, collecting packet headers for Code Red traffic on a class A network [Moore 2001] produced 0.5 Gbytes per hour of compressed NetFlow data and aggregation reduced this data only by a factor of 4. Techniques described in [Duffield et al. 2001] can be used to reduce the collection overhead at the cost of further errors. However, it can considerably *simplify* router processing to only keep track of heavy-hitters (as in our paper) if that is what the application needs.

Many papers address the problem of mapping the traffic of large IP networks. [Feldmann et al. 2000] deals with correlating measurements taken at various points to find spatial traffic distributions; the techniques in our paper can be used to complement their methods. [Duffield and Grossglauser 2000] describes a mechanism for identifying packet trajectories in the backbone, that is not focused towards estimating the traffic between various networks. [Shaikh et al. 1999] proposes that edge routers identify large long lived flows and route them along less loaded paths to achieve stable load balancing. Our algorithms might allow the detection of these candidates for re-routing in higher speed routers too.

Bloom filters [Bloom 1970] and stochastic fair blue [Feng et al. 2001] use similar but different techniques to our parallel multistage filters to compute very different metrics (set membership and drop probability). In [Tong and Reddy 1999; Smitha et al. 2001] the authors look at various mechanisms for identifying the high rate flows to ensure quality of service. Their algorithms rely on caching flow identifiers and while some of their techniques are similar to our sampling technique and to what we call preserving entries, their algorithms as a whole are quite different form ours. Gibbons and Matias [Gibbons and Matias 1998] consider synopsis data structures that use small amounts of memory to approximately summarize large databases, but their algorithms have also been used for profiling program execution[Burrows et al. 2000]. Their counting samples use the same core idea as our sample and hold algorithm. However, since the constraints and requirements in their setting (data warehouses updated constantly) are different from ours, the two final algorithms also differ. For example we need to take into account packet lengths, we operate over a sequence of measurement intervals and our algorithms need to ensure low worst per packet processing times as opposed to amortized processing in the data warehouse context. In [Fang et al. 1998], Fang et al look at efficient ways of answering *iceberg queries*, or counting the number of appearances of popular items in a database. Their multi-stage algorithm is similar to multistage filters that we propose. However, they use sampling as a front end before the filter and use multiple passes. Thus their final algorithms and analyses are very different from ours. For instance, their analysis is limited to Zipf distributions while our analysis holds for all traffic distributions. Cohen and Matias [Cohen and Matias 2003] independently discovered in the context of spectral Bloom filters the optimization to multistage filters we call conservative upsate. In [Karp et al. 2003] Karp et al. give an algorithm that is guaranteed to identify all elements that repeat frequently in a single pass. They use a second pass over the data to count exactly the number of occurrences of the frequent elements because the first pass does not guarantee accurate results. Building on our work, Narayanasamy et al. use [Narayanasamy et al. 2003] multi-stage filters with conservative update to determine execution profiles in hardware and obtain promising results.

3. OUR SOLUTION

Because our algorithms use an amount of memory that is a constant factor larger than the (relatively small) number of large flows, our algorithms can be implemented using on-chip or off-chip SRAM to store flow state. We assume that at each packet arrival we can afford to look up a flow ID in the SRAM, update the counter(s) in the entry or allocate a new entry if there is no entry associated with the current packet.

The biggest problem is to identify the large flows. Two approaches suggest themselves. First, when a packet arrives with a flow ID not in the flow memory, we could make place for the new flow by evicting the flow with the smallest measured traffic (i.e., smallest counter). While this works well on traces, it is possible to provide counter examples where a large flow is not measured because it keeps being expelled from the flow memory before its counter becomes large enough. This can happen even when using an LRU replacement policy as in [Smitha et al. 2001].

A second approach is to use classical random sampling. Random sampling (similar to sampled NetFlow except using a smaller amount of SRAM) provably identifies large flows. However, as the well known result from Table I shows, random sampling introduces a very high relative error in the measurement estimate that is proportional to $1/\sqrt{M}$, where M is the amount of SRAM used by the device. Thus one needs very high amounts of memory to reduce the inaccuracy to acceptable levels.

The two most important contributions of this paper are two new algorithms for identifying large flows: *Sample and Hold* (Section 3.1) and *Multistage Filters* (Section 3.2). Their performance is very similar, the main advantage of sample and hold being implementation simplicity, and the main advantage of multistage filters being higher accuracy. In contrast to random sampling, the relative errors of our two new algorithms scale with $1/M$, where M is the amount of SRAM. This allows our algorithms to provide much more accurate estimates than random sampling using the same amount of memory. However, unlike sampled NetFlow, our algorithms access the memory for each packet, so they must use memories fast enough to keep up with line speeds. In Section 3.3 we present improvements that further increase the accuracy of these algorithms on traces (Section 7). We start by describing the main ideas behind these schemes.

3.1 Sample and hold

Base Idea: The simplest way to identify large flows is through sampling but with the following twist. As with ordinary sampling, we sample each packet with a probability. If a packet is sampled and the flow it belongs to has no entry in the flow memory, a new entry is created. However, after an entry is created for a flow, unlike in sampled NetFlow, we update the entry for **every** subsequent packet

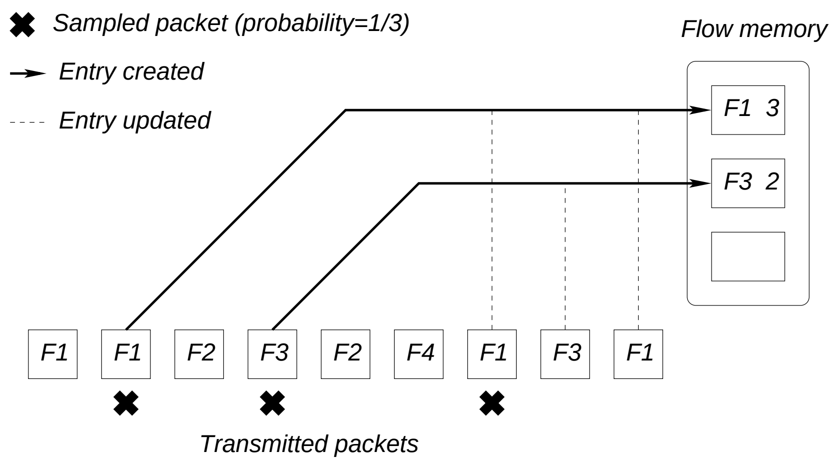

Fig. 1. The leftmost packet with flow label $F1$ arrives first at the router. After an entry is created for a flow (solid line) the counter is updated for all its packets (dotted lines)

belonging to the flow as shown in Figure 1. The counting samples of Gibbons and Matias [Gibbons and Matias 1998] use the same core idea.

Thus once a flow is *sampled*, a corresponding counter is *held* in a hash table in flow memory till the end of the measurement interval. While this clearly requires processing (looking up the flow entry and updating a counter) for every packet (unlike Sampled NetFlow), we will show that the reduced memory requirements allow the flow memory to be in SRAM instead of DRAM. This in turn allows the per-packet processing to scale with line speeds.

Let p be the probability with which we sample a byte. Thus the sampling probability for a packet of size s is $p_s = 1 - (1-p)^s \approx 1 - e^{-sp}$. This can be looked up in a precomputed table or approximated by $p_s = p * s$ (for example for packets of up to 1500 bytes and $p \leq 10^{-5}$ this approximation introduces errors smaller than 0.76% in p_s). Choosing a high enough value for p guarantees that flows above the threshold are very likely to be detected. Increasing p unduly can cause too many false positives (small flows filling up the flow memory). The advantage of this scheme is that it is easy to implement and yet gives accurate measurements with very high probability.

Preliminary Analysis: The following example illustrates the method and analysis. Suppose we wish to measure the traffic sent by flows that take over 1% of the link capacity in a measurement interval. There are at most 100 such flows. Instead of making our flow memory have just 100 locations, we will allow oversampling by a factor of 100 and keep $10,000$ locations. We wish to sample each byte with probability p such that the average number of samples is $10,000$. Thus if C bytes can be transmitted in the measurement interval, $p = 10,000/C$.

For the error analysis, consider a flow F that takes 1% of the traffic. Thus F sends more than $C/100$ bytes. Since we are randomly sampling each byte with probability $10,000/C$, the probability that F will not be in the flow memory at the

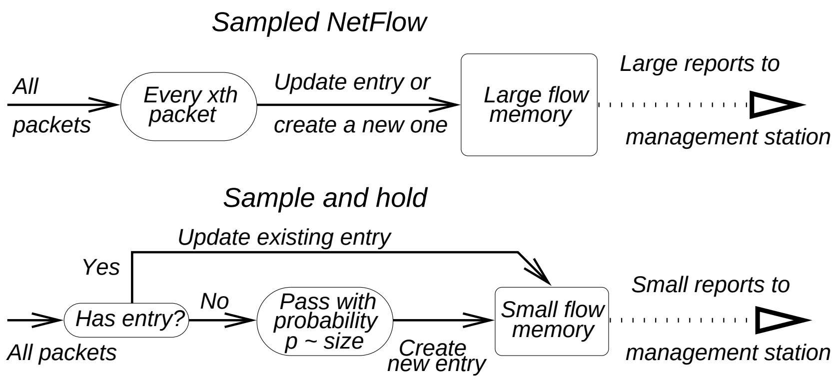

Fig. 2. Sampled NetFlow counts only sampled packets, sample and hold counts all after entry created

end of the measurement interval (false negative) is $(1-10000/C)^{C/100}$ which is very close to e^{-100}. Notice that the factor of 100 in the exponent is the oversampling factor. Better still, the probability that flow F is in the flow memory after sending 5% of its traffic is, similarly, $1-e^{-5}$ which is greater than 99% probability. Thus with 99% probability the reported traffic for flow F will be at most 5% below the actual amount sent by F.

The analysis can be generalized to arbitrary threshold values; the memory needs to scale inversely with the threshold percentage and directly with the oversampling factor. Notice also that the analysis assumes that there is always space to place a sample flow not already in the memory. Setting $p = 10,000/C$ ensures only that the *average* number of flows sampled[3] is no more than 10,000. However, the distribution of the number of samples is binomial with a small standard deviation (square root of the mean). Thus, adding a few standard deviations to the memory estimate (e.g., a total memory size of 10,300) makes it extremely unlikely that the flow memory will ever overflow[4].

Compared to Sampled NetFlow our idea has three significant differences. Most importantly, we sample only to decide whether to add a flow to the memory; from that point on, we update the flow memory with every byte the flow sends as shown in Figure 2. As Section 5 shows this will make our results much more accurate. Second, our sampling technique avoids packet size biases unlike NetFlow which samples every x packets. Third, our technique reduces the extra resource overhead (router processing, router memory, network bandwidth) for sending large reports

[3]Our analyses from Section 4.1 and from [Estan and Varghese 2002] also give tight upper bounds on the number of entries used that hold with high probability.

[4]If the flow memory overflows, we can not create new entries until entries are freed at the beginning of the next measurement interval and thus large flows might go undetected. Allocating more memory is probably not an option for hardware implementations. Selectively discarding the least important entries requires us to traverse the entire flow memory and this would violate the strict bounds we have for per packet processing time.

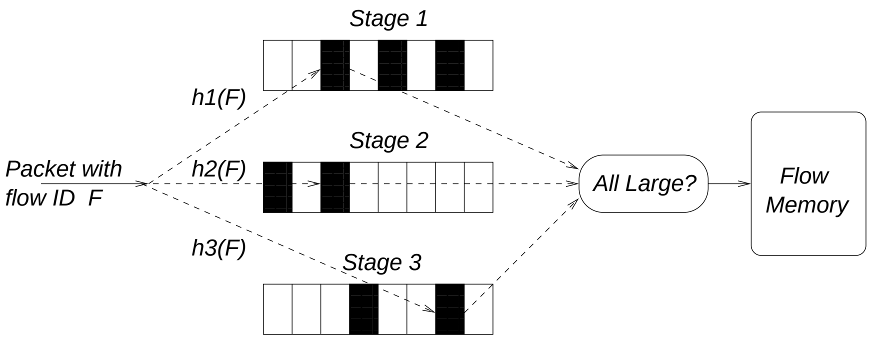

Fig. 3. In a parallel multistage filter, a packet with a flow ID F is hashed using hash function $h1$ into a Stage 1 table, $h2$ into a Stage 2 table, etc. Each table entry contains a counter that is incremented by the packet size. If *all* the hashed counters are above the threshold (shown bolded), F is passed to the flow memory for individual observation.

with many records to a management station.

3.2 Multistage filters

Base Idea: The basic multistage filter is shown in Figure 3. The building blocks are hash stages that operate in parallel. First, consider how the filter operates with only one stage. A stage is a table of counters which is indexed by a hash function computed on a packet flow ID; all counters in the table are initialized to 0 at the start of a measurement interval. When a packet comes in, a hash on its flow ID is computed and the size of the packet is added to the corresponding counter. Since all packets belonging to the same flow hash to the same counter, if a flow F sends more than threshold T, F's counter will exceed the threshold. If we add to the flow memory all packets that hash to counters of T or more, we are guaranteed to identify all the large flows (no false negatives). The multi-stage algorithm of Fang et al [Fang et al. 1998] is similar to our multistage filters and the accounting bins of stochastic fair blue [Feng et al. 2001] use a similar data structure to compute drop probabilities for active queue management.

Unfortunately, since the number of counters we can afford is significantly smaller than the number of flows, many flows will map to the same counter. This can cause false positives in two ways: first, small flows can map to counters that hold large flows and get added to flow memory; second, several small flows can hash to the same counter and add up to a number larger than the threshold.

To reduce this large number of false positives, we use multiple stages. Each stage (Figure 3) uses an *independent* hash function. Only the packets that map to counters of T or more at *all* stages get added to the flow memory. For example, in Figure 3, if a packet with a flow ID F arrives that hashes to counters 3, 3, and 7 respectively at the three stages, F will pass the filter (counters that are over the threshold are shown darkened). On the other hand, a flow G that hashes to counters 7, 5, and 4 will not pass the filter because the second stage counter is not over the threshold. Effectively, the multiple stages attenuate the probability of false positives exponentially in the number of stages. This is shown by the following simple analysis.

Preliminary Analysis: Assume a 100 Mbytes/s link[5], with 100,000 flows and we want to identify the flows above 1% of the link during a one second measurement interval. Assume each stage has 1,000 buckets and a threshold of 1 Mbyte. Let's see what the probability is for a flow sending 100 Kbytes to pass the filter. For this flow to pass one stage, the other flows need to add up to 1 Mbyte - 100 Kbytes = 900 Kbytes. There are at most 99,900/900=111 such buckets out of the 1,000 at each stage. Therefore, the probability of passing one stage is at most 11.1%. With 4 independent stages, the probability that a certain flow no larger than 100 Kbytes passes all 4 stages is the *product* of the individual stage probabilities which is at most $1.52 * 10^{-4}$.

Based on this analysis, we can dimension the flow memory so that it is large enough to accommodate all flows that pass the filter. The expected number of flows below 100 Kbytes passing the filter is at most $100,000 * 15.2 * 10^{-4} < 16$. There can be at most 999 flows above 100 Kbytes, so the number of entries we expect to accommodate all flows is at most 1,015. Section 4 has a rigorous theorem that proves a stronger bound (for this example 122 entries) that holds for any distribution of flow sizes. Note the potential scalability of the scheme. If the number of flows increases to 1 million, we simply add a fifth hash stage to get the same effect. Thus to handle 100,000 flows requires roughly 4000 counters and a flow memory of approximately 100 memory locations, while to handle 1 million flows requires roughly 5000 counters and the same size of flow memory. This is logarithmic scaling.

The number of memory accesses per packet for a multistage filter is one read and one write per stage. If the number of stages is small, this is feasible even at high speeds by doing parallel memory accesses to each stage in a chip implementation.[6] Multistage filters also need to compute the hash functions. These can be computed efficiently in hardware. For software implementations this adds to the per packet processing and can replace memory accesses as the main bottleneck. However, we already need to compute a hash function to locate the per flow entries in the flow memory, thus one can argue that we do not introduce a new problem, just make an existing one worse. While multistage filters are more complex than sample-and-hold, they have two important advantages. They reduce the probability of false negatives to 0 and decrease the probability of false positives, thereby reducing the size of the required flow memory.

3.2.1 *The serial multistage filter.* We briefly present a variant of the multistage filter called a serial multistage filter (Figure 4). Instead of using multiple stages in parallel, we can place them serially after each other, each stage seeing only the packets that passed the previous stage.

Let d be the number of stages (the depth of the serial filter). We set a *stage threshold* of T/d for all the stages. Thus for a flow that sends T bytes, by the time the last packet is sent, the counters the flow hashes to at all d stages reach T/d, so the packet will pass to the flow memory. As with parallel filters, we have no false

[5] To simplify computation, in our examples we assume that 1Mbyte=1,000,000 bytes and 1Kbyte=1,000 bytes.
[6] We describe details of a preliminary OC-192 chip implementation of multistage filters in Section 8.

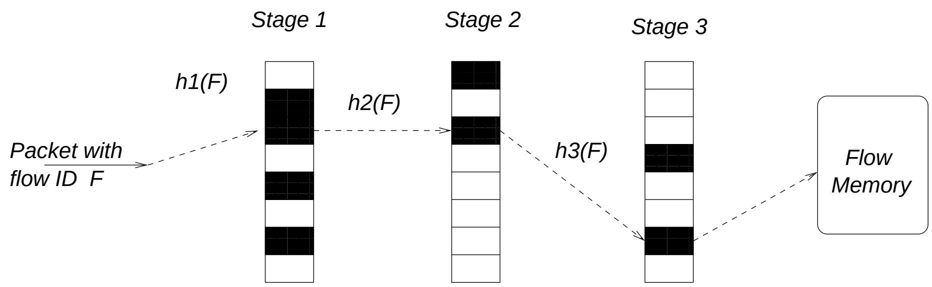

Fig. 4. In a serial multistage filter, a packet with a flow ID F is hashed using hash function $h1$ into a Stage 1 table. If the counter is below the stage threshold T/d, it is incremented. If the counter reaches the stage threshold the packet is hashed using function $h2$ to a Stage 2 counter, etc. If the packet passes all stages, an entry is created for F in the flow memory.

negatives. As with parallel filters, small flows can pass the filter only if they keep hashing to counters made large by other flows.

The analytical evaluation of serial filters is more complicated than for parallel filters. On one hand the early stages shield later stages from much of the traffic, and this contributes to stronger filtering. On the other hand the threshold used by stages is smaller (by a factor of d) and this contributes to weaker filtering. Since, as shown in Section 7, parallel filters perform better than serial filters on traces of actual traffic, the main focus in this paper will be on parallel filters.

3.3 Improvements to the basic algorithms

The improvements to our algorithms presented in this section further increase the accuracy of the measurements and reduce the memory requirements. Some of the improvements apply to both algorithms, some apply only to one of them.

3.3.1 *Basic optimizations.* There are a number of basic optimizations that exploit the fact that large flows often last for more than one measurement interval.

Preserving entries: Erasing the flow memory after each interval implies that the bytes of a large flow sent before the flow is allocated an entry are not counted. By preserving entries of large flows across measurement intervals and only reinitializing stage counters, *all long lived large flows are measured nearly exactly.* To distinguish between a large flow that was identified late and a small flow that was identified by error, a conservative solution is to preserve the entries of not only the flows for which we count at least T bytes in the current interval, but also all the flows that were added in the current interval (since they may be large flows that entered late).

Early removal: Sample and hold has a larger rate of false positives than multistage filters. If we keep for one more interval all the flows that obtained a new entry, many small flows will keep their entries for two intervals. We can improve the situation by selectively removing some of the flow entries created in the current interval. The new rule for preserving entries is as follows. We define an early removal threshold R that is less then the threshold T. At the end of the measurement interval, we keep all entries whose counter is at least T and all entries that have been added during the current interval and whose counter is at least R.

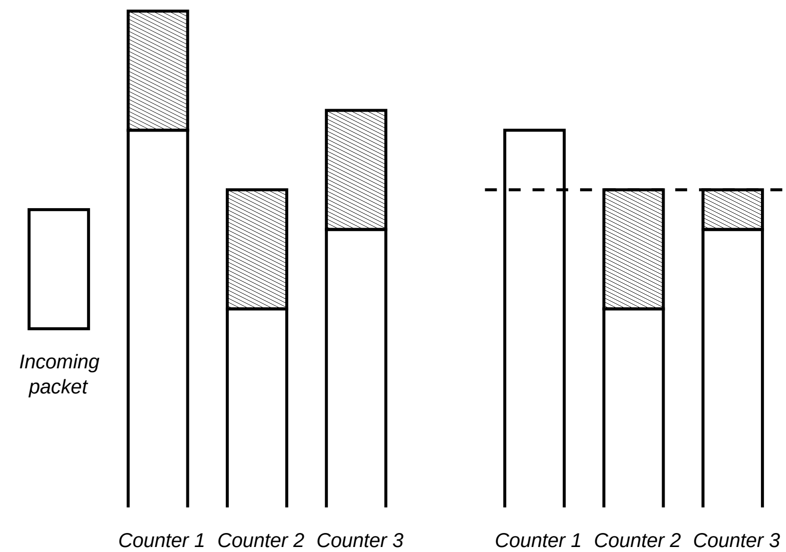

Fig. 5. Conservative update: without conservative update (left) all counters are increased by the size of the incoming packet, with conservative update (right) no counter is increased to more than the size of the smallest counter plus the size of the packet

Shielding: Consider large, long lived flows that go through the filter each measurement interval. Each measurement interval, the counters they hash to exceed the threshold. With shielding, traffic belonging to flows that have an entry in flow memory no longer passes through the filter (the counters in the filter are not incremented for packets with an entry), thereby reducing false positives. If we shield the filter from a large flow, many of the counters it hashes to will not reach the threshold after the first interval. This reduces the probability that a random small flow will pass the filter by hashing to counters that are large because of other flows.

3.3.2 *Conservative update of counters.* We now describe an important optimization for multistage filters that improves performance by an order of magnitude. *Conservative update* reduces the number of false positives of multistage filters by three subtle changes to the rules for updating counters. In essence, we endeavour to increment counters as little as possible (thereby reducing false positives by preventing small flows from passing the filter) while still avoiding false negatives (i.e., we need to ensure that all flows that reach the threshold still pass the filter.)

The first change (Figure 5) applies only to parallel filters and only for packets that don't pass the filter. As usual, an arriving flow F is hashed to a counter at each stage. We update the smallest of the counters normally (by adding the size of the packet). *However, the other counters are set to the maximum of their old value and the new value of the smallest counter.* Since the amount of traffic sent by the current flow is at most the new value of the smallest counter, this change *cannot introduce a false negative* for the flow the packet belongs to. Since we never decrement counters, other large flows that might hash to the same counters are not prevented from passing the filter.

The second change is very simple and applies to both parallel and serial filters. When a packet passes the filter and it obtains an entry in the flow memory, no counters should be updated. This will leave the counters below the threshold. Other flows with smaller packets that hash to these counters will get less "help" in passing the filter.

The third change applies only to serial filters. It regards the way counters are updated when the threshold is exceeded in any stage but the last one. Let's say the value of the counter a packet hashes to at stage i is $T/d-x$ and the size of the packet is $s > x > 0$. Normally one would increment the counter at stage i to T/d and add $s-x$ to the counter from stage $i+1$. What we can do instead with the counter at stage $i+1$ is update its value to the maximum of $s-x$ and its old value (assuming $s-x < T/d$). Since the counter at stage i was below T/d, we know that no prior packets belonging to the same flow as the current one passed this stage and contributed to the value of the counter at stage $i+1$. We could not apply this change if the threshold T was allowed to change during a measurement interval.

4. ANALYTICAL EVALUATION OF OUR ALGORITHMS

In this section we analytically evaluate our algorithms. We only present the main results. The proofs, supporting lemmas and some of the less important results (e.g. high probability bounds corresponding to our bounds on the average number of flows passing a multistage filter) are in [Estan and Varghese 2002]. We focus on two important questions:

—*How good are the results?* We use two distinct measures of the quality of the results: how many of the large flows are identified, and how accurately is their traffic estimated?

—*What are the resources required by the algorithm?* The key resource measure is the size of flow memory needed. A second resource measure is the number of memory references required.

In Section 4.1 we analyze our sample and hold algorithm, and in Section 4.2 we analyze multistage filters. We first analyze the basic algorithms and then examine the effect of some of the improvements presented in Section 3.3. In the next section (Section 5) we use the results of this section to analytically compare our algorithms with sampled NetFlow.

Example: We will use the following running example to give numeric instances. Assume a 100 Mbyte/s link with $100,000$ flows. We want to measure all flows whose traffic is more than 1% (1 Mbyte) of link capacity in a one second measurement interval.

4.1 Sample and hold

We first define some notation we use in this section.

—p the probability for sampling a byte;

—s the size of a flow (in bytes);

—T the threshold for large flows;

—C the capacity of the link – the number of bytes that can be sent during the *entire* measurement interval;

—O the oversampling factor defined by $p = O \cdot 1/T$;

—c the number of bytes actually counted for a flow.

4.1.1 *The quality of results for sample and hold.* The first measure of the quality of the results is the probability that a flow at the threshold is not identified. As presented in Section 3.1 the probability that a flow of size T is not identified is $(1-p)^T \approx e^{-O}$. An oversampling factor of 20 results in a probability of missing flows at the threshold of $2 * 10^{-9}$.

Example: For our example, p must be 1 in 50,000 bytes for an oversampling of 20. With an average packet size of 500 bytes this is roughly 1 in 100 packets.

The second measure of the quality of the results is the difference between the size of a flow s and our estimate. The number of bytes that go by before the first one gets sampled has a geometric probability distribution[7]: it is x with a probability[8] $(1-p)^x p$.

Therefore $E[s-c] = 1/p$ and $SD[s-c] = \sqrt{1-p}/p$. The best estimate for s is $c+1/p$ and its standard deviation is $\sqrt{1-p}/p$. If we choose to use c as an estimate for s then the error will be larger, but we never overestimate the size of the flow[9]. In this case, the deviation from the actual value of s is $\sqrt{E[(s-c)^2]} = \sqrt{2-p}/p$. Based on this value we can also compute the relative error of a flow of size T which is $T\sqrt{2-p}/p = \sqrt{2-p}/O$.

Example: For our example, with an oversampling factor O of 20, the relative error (computed as the standard deviation of the estimate divided by the actual value) for a flow at the threshold is 7%.

4.1.2 *The memory requirements for sample and hold.* The size of the flow memory is determined by the number of flows identified. The actual number of sampled packets is an upper bound on the number of entries needed in the flow memory because new entries are created only for sampled packets. Assuming that the link is constantly busy, by the linearity of expectation, the expected number of sampled bytes is $p \cdot C = O \cdot C/T$.

Example: Using an oversampling of 20 requires 2,000 entries on average.

The number of sampled bytes can exceed this value. Since the number of sampled bytes has a binomial distribution, we can use the normal curve to bound with high probability the number of bytes sampled during the measurement interval. Therefore with probability 99% the actual number will be at most 2.33 standard deviations above the expected value; similarly, with probability 99.9% it will be at most 3.08 standard deviations above the expected value. The standard deviation of the number of sampled bytes is $\sqrt{Cp(1-p)}$.

Example: For an oversampling of 20 and an overflow probability of 0.1% we need at most 2,147 entries.

---

[7]We ignore for simplicity that the bytes before the first sampled byte that are in the same packet with it are also counted. Therefore the actual algorithm will be more accurate than this model.
[8]Since we focus on large flows, we ignore for simplicity the correction factor we need to apply to account for the case when the flow goes undetected (i.e. x is actually bound by the size of the flow s, but we ignore this).
[9]Gibbons and Matias [Gibbons and Matias 1998] have a more elaborate analysis and use a different correction factor.

This result can be further tightened if we make assumptions about the distribution of flow sizes and thus account for very large flows having many of their packets sampled. Let's assume that the flows have a Zipf (Pareto) distribution with parameter 1 defined as $Prs > x = constant * x^{-1}$. If we have n flows that use the whole bandwidth C, the total traffic of the largest j flows is at least $C\frac{ln(j+1)}{ln(2n+1)}$[Estan and Varghese 2002]. For any value of j between 0 and n we obtain an upper bound on the number of entries expected to be used in the flow memory by assuming that the largest j flows always have an entry by having at least one of their packets sampled and each packet sampled from the rest of the traffic creates an entry: $j + Cp(1 - ln(j+1)/ln(2n+1)$. By differentiating we obtain the value of j that provides the tightest bound: $j = Cp/ln(2n+1) - 1$.

Example: Using an oversampling of 20 requires at most 1,328 entries on average.

4.1.3 *The effect of preserving entries.* We preserve entries across measurement intervals to improve accuracy. The probability of missing a large flow decreases because we cannot miss it if we keep its entry from the prior interval. Accuracy increases because we know the exact size of the flows whose entries we keep. To quantify these improvements we need to know the ratio of long lived flows among the large ones.

The cost of this improvement in accuracy is an increase in the size of the flow memory. We need enough memory to hold the samples from both measurement intervals[10]. Therefore the expected number of entries is bounded by $2O \cdot C/T$.

To bound with high probability the number of entries we use the normal curve and the standard deviation of the number of sampled packets during the 2 intervals which is $\sqrt{2Cp(1-p)}$.

Example: For an oversampling of 20 and acceptable probability of overflow equal to 0.1%, the flow memory has to have at most 4,207 entries to preserve entries.

4.1.4 *The effect of early removal.* The effect of early removal on the proportion of false negatives depends on whether or not the entries removed early are reported. Since we believe it is more realistic that implementations will not report these entries, we will use this assumption in our analysis. Let $R < T$ be the early removal threshold. A flow at the threshold is not reported unless one of its first $T - R$ bytes is sampled. Therefore the probability of missing the flow is approximately $e^{-O(T-R)/T}$. If we use an early removal threshold of $R = 0.2 * T$, this increases the probability of missing a large flow from $2 * 10^{-9}$ to $1.1 * 10^{-7}$ with an oversampling of 20.

Early removal reduces the size of the memory required by limiting the number of entries that are preserved from the previous measurement interval. Since there can be at most C/R flows sending R bytes, the number of entries that we keep is at most C/R which can be smaller than OC/T, the bound on the expected number of sampled packets. The expected number of entries we need is $C/R + OC/T$.

To bound with high probability the number of entries we use the normal curve. If $R \geq T/O$ the standard deviation is given only by the randomness of the packets

[10]We actually also keep the older entries that are above the threshold. Since we are performing a worst case analysis we assume that there is no flow above the threshold, because if there were, many of its packets would be sampled, decreasing the number of entries required.

sampled in one interval and is $\sqrt{Cp(1-p)}$.

Example: An oversampling of 20 and $R = 0.2T$ with overflow probability 0.1% requires 2,647 memory entries.

4.2 Multistage filters

In this section, we analyze parallel multistage filters. We first define some new notation:

—b the number of buckets in a stage;

—d the depth of the filter (the number of stages);

—n the number of active flows;

—k the stage strength is the ratio of the threshold and the average size of a counter. $k = \frac{T\ b}{C}$, where C denotes the channel capacity as before. Intuitively, this is the factor we inflate each stage memory beyond the minimum of C/T .

Example: To illustrate our results numerically, we will assume that we solve the measurement example described in Section 4 with a 4 stage filter, with 1000 buckets at each stage. The stage strength k is 10 because each stage memory has 10 times more buckets than the maximum number of flows (i.e., 100) that can cross the specified threshold of 1%.

4.2.1 *The quality of results for multistage filters.* As discussed in Section 3.2, multistage filters have no false negatives. The error of the traffic estimates for large flows is bounded by the threshold T since no flow can send T bytes without being entered into the flow memory. The stronger the filter, the less likely it is that the flow will be entered into the flow memory much before it reaches T. We first state an upper bound for the probability of a small flow passing the filter described in Section 3.2.

LEMMA 1. *Assuming the hash functions used by different stages are independent, the probability of a flow of size $s < T(1-1/k)$ passing a parallel multistage filter is at most $p_s \leq \left(\frac{1}{k}\frac{T}{T-s}\right)^d$.*

The proof of this bound formalizes the preliminary analysis of multistage filters from Section 3.2. Note that the bound *makes no assumption about the distribution of flow sizes*, and thus applies for all flow distributions. We only assume that the hash functions are random and independent. The bound is tight in the sense that it is almost exact for a distribution that has $\lfloor (C-s)/(T-s) \rfloor$ flows of size $(T-s)$ that send all their packets before the flow of size s. However, for realistic traffic mixes (e.g., if flow sizes follow a Zipf distribution), this is a very conservative bound.

Based on this lemma we obtain a lower bound for the expected error for a large flow.

THEOREM 2. *The expected number of bytes of a large flow of size s undetected by a multistage filter is bound from below by*

$$E[s-c] \geq T\left(1 - \frac{d}{k(d-1)}\right) - y_{max} \tag{1}$$

where y_{max} is the maximum size of a packet.

This bound suggests that we can significantly improve the accuracy of the estimates by adding a correction factor to the bytes actually counted. The down side to adding a correction factor is that we can overestimate some flow sizes; this may be a problem for accounting applications. The y_{max} factor from the result comes from the fact that when the packet that makes the counters exceed the threshold arrives, c is initialized to its size which can be as much as y_{max}.

4.2.2 *The memory requirements for multistage filters.* We can dimension the flow memory based on bounds on the number of flows that pass the filter. Based on Lemma 1 we can compute a bound on the total number of flows expected to pass the filter (the full derivation of this theorem is in Appendix A).

THEOREM 3. *The expected number of flows passing a parallel multistage filter is bound by*

$$E[n_{pass}] \leq max\left(\frac{b}{k-1}, n\left(\frac{n}{kn-b}\right)^d\right) + n\left(\frac{n}{kn-b}\right)^d \tag{2}$$

Example: Theorem 3 gives a bound of 121.2 flows. Using 3 stages would have resulted in a bound of 200.6 and using 5 would give 112.1. Note that when the first term dominates the max, there is not much gain in adding more stages.

We can also bound the number of flows passing the filter with high probability.

Example: The probability that more than 185 flows pass the filter is at most 0.1%. Thus by increasing the flow memory from the expected size of 122 to 185 we can make overflow of the flow memory extremely improbable.

As with sample and hold, making assumptions about the distribution of flow sizes can lead to a smaller bound on the number of flows expected to enter the flow memory[Estan and Varghese 2002].

THEOREM 4. *If the flows sizes have a Zipf distribution with parameter 1, the expected number of flows passing a parallel multistage filter is bound by*

$$E[n_{pass}] \leq i_0 + \frac{n}{k^d} + \frac{db}{k^{d+1}} + \frac{db\ ln(n+1)^{d-2}}{k^2\left(k\ ln(n+1) - \frac{b}{i_0-0.5}\right)^{d-1}} \tag{3}$$

where $i_0 = \lceil max(1.5 + \frac{b}{k\ ln(n+1)}, \frac{b}{ln(2n+1)(k-1)})\rceil$.

Example: Theorem 4 gives a bound of 21.7 on the number of flows expected to pass the filter.

4.2.3 *The effect of preserving entries and shielding.* Preserving entries affects the accuracy of the results the same way as for sample and hold: long lived large flows have their traffic counted exactly after their first interval above the threshold. As with sample and hold, preserving entries basically doubles all the bounds for memory usage.

Shielding has a strong effect on filter performance, since it reduces the traffic presented to the filter. Reducing the traffic α times increases the stage strength to $k * \alpha$, which can be substituted in Theorems 2 and 3.

Measure	Sample and hold	Multistage filters	Sampling
Relative error for a flow of size zC	$\frac{\sqrt{2}}{Mz}$	$\frac{1+10\ r\ log_{10}(n)}{Mz}$	$\frac{1}{\sqrt{Mz}}$
Memory accesses per packet	1	$1 + log_{10}(n)$	$\frac{1}{x} = \frac{M}{C}$

Table I. Comparison of the core algorithms: sample and hold provides most accurate results while pure sampling has very few memory accesses

5. COMPARING MEASUREMENT METHODS

In this section we analytically compare the performance of three traffic measurement algorithms: our two new algorithms (sample and hold and multistage filters) and Sampled NetFlow. First, in Section 5.1, we compare the algorithms at the core of traffic measurement devices. For the core comparison, we assume that each of the algorithms is given the *same* amount of high speed memory and we compare their accuracy and number of memory accesses. This allows a fundamental analytical comparison of the effectiveness of each algorithm in identifying heavy-hitters.

However, in practice, it may be unfair to compare Sampled NetFlow with our algorithms using the same amount of memory. This is because Sampled NetFlow can afford to use a large amount of DRAM (because it does not process every packet) while our algorithms cannot (because they process every packet and hence need to store per flow entries in SRAM). Thus we perform a second comparison in Section 5.2 of complete traffic measurement devices. In this second comparison, we allow Sampled NetFlow to use more memory than our algorithms. The comparisons are based on the algorithm analysis in Section 4 and an analysis of NetFlow taken from [Estan and Varghese 2002].

5.1 Comparison of the core algorithms

In this section we compare sample and hold, multistage filters and ordinary sampling (used by NetFlow) under the assumption that they are all constrained to using M memory entries. More precisely, the expected number of memory entries used is at most M irrespective of the distribution of flow sizes. We focus on the accuracy of the measurement of a flow (defined as the standard deviation of an estimate over the actual size of the flow) whose traffic is zC (for flows of 1% of the link capacity we would use $z = 0.01$).

The bound on the expected number of entries is the same for sample and hold and for sampling and is pC. By making this equal to M we can solve for p. By substituting in the formulae we have for the accuracy of the estimates and after eliminating some terms that become insignificant (as p decreases and as the link capacity goes up) we obtain the results shown in Table I.

For multistage filters, we use a simplified version of the result from Theorem 3: $E[n_{pass}] \leq b/k + n/k^d$. We increase the number of stages used by the multistage filter logarithmically as the number of flows increases so that only a single small flow is expected to pass the filter[11] and the strength of the stages is 10. At this point we

[11] Configuring the filter such that a small number of small flows pass would have resulted in smaller memory and fewer memory accesses (because we would need fewer stages), but it would have complicated the formulae.

Measure	Sample and hold	Multistage filters	Sampled NetFlow
Exact measurements	$\lessapprox$ $longlived\%$	$longlived\%$	0
Relative error	$1.41/O$	$\lessapprox$ $1/u$	$0.0088/\sqrt{zt}$
Memory bound	$2O/z$	$2/z + 1/z\ log_{10}(n)$	min(n,486000 t)
Memory accesses	1	$1 + log_{10}(n)$	$1/x$

Table II. Comparison of traffic measurement devices

estimate the memory usage to be $M = b/k + 1 + rbd = C/T + 1 + r10log_{10}(n)C/T$ where $r < 1$ depends on the implementation and reflects the relative cost of a counter and an entry in the flow memory. From here we obtain T which will be an upper bound on the error of our estimate of flows of size zC. From here, the result from Table I is immediate.

The term Mz that appears in all formulae in the first row of the table is exactly equal to the oversampling we defined in the case of sample and hold. It expresses how many times we are willing to allocate over the theoretical minimum memory to obtain better accuracy. We can see that the error of our algorithms decreases inversely proportional to this term while the error of sampling is proportional to the inverse of its square root.

The second line of Table I gives the number of memory locations accessed per packet by each algorithm. Since sample and hold performs a packet lookup for every packet[12], its per packet processing is 1. Multistage filters add to the one flow memory lookup an extra access to one counter per stage and the number of stages increases as the logarithm of the number of flows. Finally, for ordinary sampling one in $x = C/M$ packets get sampled so the average per packet processing is $1/x = M/C$.

Table I provides a fundamental comparison of our new algorithms with ordinary sampling as used in Sampled NetFlow. The first line shows that the relative error of our algorithms scales with $1/M$ which is much better than the $1/\sqrt{M}$ scaling of ordinary sampling. However, the second line shows that this improvement comes at the cost of requiring at least one memory access per packet for our algorithms. While this allows us to implement the new algorithms using SRAM, the smaller number of memory accesses ($\ll 1$) per packet allows Sampled NetFlow to use DRAM. This is true as long as x is larger than the ratio of a DRAM memory access to an SRAM memory access. However, even a DRAM implementation of Sampled NetFlow has some problems which we turn to in our second comparison.

5.2 Comparing Measurement Devices

Table I implies that increasing DRAM memory size M to infinity can reduce the relative error of Sampled NetFlow to zero. But this assumes that by increasing memory one can increase the sampling rate so that x becomes arbitrarily close to 1. If $x = 1$, there would be no error since every packet is logged. But x must at

[12]We equate a lookup in the flow memory to a single memory access. This is true if we use a content associable memory. Lookups without hardware support require a few more memory accesses to resolve hash collisions.

least be as large as the ratio of DRAM speed (currently around 60 ns) to SRAM speed (currently around 5 ns); thus Sampled NetFlow will always have a minimum error corresponding to this value of x even when given unlimited DRAM.

With this insight, we now compare the performance of our algorithms and NetFlow in Table II without limiting NetFlow memory. Thus Table II takes into account the underlying technologies (i.e., the potential use of DRAM over SRAM) and one optimization (i.e., preserving entries) for both of our algorithms.

We consider the task of estimating the size of all the flows above a fraction z of the link capacity over a measurement interval of t seconds. In order to make the comparison possible we change somewhat the way NetFlow operates: we assume that it reports the traffic data for each flow after each measurement interval, like our algorithms do. The four characteristics of the traffic measurement algorithms presented in the table are: the percentage of large flows known to be measured exactly, the relative error of the estimate of a large flow, the upper bound on the memory size and the number of memory accesses per packet.

Note that the table does not contain the actual memory used but a bound. For example the number of entries used by NetFlow is bounded by the number of active flows and the number of DRAM memory lookups that it can perform during a measurement interval (which doesn't change as the link capacity grows)[13]. Our measurements in Section 7 show that for all three algorithms the actual memory usage is much smaller than the bounds, especially for multistage filters. Memory is measured in entries, not bytes. We assume that a flow memory entry is equivalent to 10 of the counters used by the filter (i.e. $r = 1/10$) because the flow ID is typically much larger than the counter. Note that the number of memory accesses required per packet does not necessarily translate to the time spent on the packet because memory accesses can be pipelined or performed in parallel.

We make simplifying assumptions about technology evolution. As link speeds increase, so must the electronics. Therefore we assume that SRAM speeds keep pace with link capacities. We also assume that the speed of DRAM does not improve significantly ([Patterson and Hennessy 1998] states that DRAM speeds improve only at 9% per year while clock rates improve at 40% per year).

We assume the following configurations for the three algorithms. Our algorithms preserve entries. For multistage filters we introduce a new parameter expressing how many times larger a flow of interest is than the threshold of the filter $u = zC/T$. Since the speed gap between the DRAM used by sampled NetFlow and the link speeds increases as link speeds increase, NetFlow has to decrease its sampling rate proportionally with the increase in capacity[14] to provide the smallest possible error. For the NetFlow error calculations we also assume that the size of the packets of large flows is 1500 bytes.

Besides the differences that stem from the core algorithms (Table I), we see new differences in Table II. The first big difference (Row 1 of Table II) is that unlike

[13]The limit on the number of packets NetFlow can process we used for Table II is based on Cisco documentation that states that sampling should turned on for speeds larger than OC-3 (155.52 Mbits/second). Thus we assumed that this is the ,maximum speed at which NetFlow can handle minimum sized (40 byte) packets.

[14]If the capacity of the link is x times OC-3, then one in x packets gets sampled. We assume based on [NetFlow] that NetFlow can handle packets no smaller than 40 bytes at OC-3 speeds.

NetFlow, *our algorithms provide exact measures for long-lived large flows* by preserving entries. More precisely, by preserving entries our algorithms will exactly measure traffic for all (or almost all in the case of sample and hold) of the large flows that were large in the previous interval. Given that our measurements show that most large flows are long lived (depending on the flow definition, the average percentage of the large flows that were large in the previous measurement interval is between 56% and 81%), this is a big advantage.

Of course, one could get the same advantage by using an SRAM flow memory that preserves large flows across measurement intervals in Sampled NetFlow as well. However, that would require the router to root through its DRAM flow memory before the end of the interval to find the large flows, a large processing load. One can also argue that if one can afford an SRAM flow memory, it is quite easy to do sample and hold.

The second big difference (Row 2 of Table II) is that we can make our algorithms arbitrarily accurate at the cost of increases in the amount of memory used[15] while sampled NetFlow can do so only by increasing the measurement interval t.

The third row of Table II compares the memory used by the algorithms. The extra factor of 2 for sample and hold and multistage filters arises from preserving entries. Note that the number of entries used by Sampled NetFlow is bounded by both the number n of active flows and the number of memory accesses that can be made in t seconds. Finally, the fourth row of Table II is identical to the second row of Table I.

Table II demonstrates that our algorithms have two advantages over NetFlow: **i)** they provide exact values for long-lived large flows (row 1) and **ii)** they provide much better accuracy even for small measurement intervals (row 2). Besides these advantages, our algorithms also have three more advantages not shown in Table II. These are **iii)** provable lower bounds on traffic, **iv)** reduced resource consumption for collection, and **v)** faster detection of new large flows. We now examine advantages **iii)** through **v)** in more detail.

iii) Provable Lower Bounds: A possible disadvantage of Sampled NetFlow is that the NetFlow estimate is not an actual lower bound on the flow size. Thus a customer may be charged for more than the customer sends. While one can make the probability of overcharging arbitrarily low (using large measurement intervals or other methods from [Duffield et al. 2001]), there may be philosophical objections to overcharging. Our algorithms do not have this problem.

iv) Reduced Resource Consumption: Clearly, while Sampled NetFlow can increase DRAM to improve accuracy, the router has more entries at the end of the measurement interval. These records have to be processed, potentially aggregated, and transmitted over the network to the management station. If the router extracts the heavy hitters from the log, then router processing is large; if not, the bandwidth consumed and processing at the management station is large. By using fewer entries, our algorithms avoid these resource (e.g., memory, transmission bandwidth, and router CPU cycles) bottlenecks, but as detailed in Table II sample and hold and multistage filters incur more upfront work by processing each packet.

[15]Of course, technology and cost impose limitations on the amount of available SRAM but the current limits for on and off-chip SRAM are high enough for our algorithms.

```
ADAPTTHRESHOLD
  usage = entriesused/flowmemsize
  if (usage > target)
    threshold = threshold * (usage/target)^adjustup
  else
    if (threshold did not increase for 3 intervals)
      threshold = threshold * (usage/target)^adjustdown
    endif
  endif
```

Fig. 6. Dynamic threshold adaptation to achieve target memory usage

6. DIMENSIONING TRAFFIC MEASUREMENT DEVICES

We describe how to dimension our algorithms. For applications that face adversarial behavior (e.g., detecting DoS attacks), one should use the conservative bounds from Sections 4.1 and 4.2. Other applications such as accounting can obtain greater accuracy from more aggressive dimensioning as described below. The measurements from Section 7 show that the gains can be substantial. For example the number of false positives for a multistage filter can be four orders of magnitude below what the conservative analysis predicts. To avoid a priori knowledge of flow distributions, we adapt algorithm parameters to actual traffic. The main idea is to *keep decreasing the threshold below the conservative estimate until the flow memory is nearly full* (totally filling the memory can result in new large flows not being tracked).

Dynamically adapting the threshold is an effective way to control memory usage. Sampled NetFlow uses a fixed sampling rate that is either so low that a small percentage of the memory is used all or most of the time, or so high that the memory is filled and NetFlow is forced to expire entries which might lead to inaccurate results exactly when they are most important: when the traffic surges.

Figure 6 presents our threshold adaptation algorithm. There are two important constants that adapt the threshold to the traffic: the "target usage" (variable *target* in Figure 6) that tells it how full the memory can be without risking filling it up completely and the "adjustment ratio" (variables *adjustup* and *adjustdown* in Figure 6) that the algorithm uses to decide how much to adjust the threshold to achieve a desired increase or decrease in flow memory usage. To give stability to the traffic measurement device, the *entriesused* variable does not contain the number of entries used over the last measurement interval, but an average of the last 3 intervals.

We use measurements (presented in [Estan and Varghese 2002]) to find good values for the target usage and the adjustment ratio. We want the target usage as high as possible, but still low enough so that the short-term fluctuations in the number of large flows do not cause the flow memory to fill up. Based on measurements, we use a target memory usage of 90% in our experiments. The adjustment ratio reflects how our traffic measurement device adapts to longer term fluctuations in the number of large flows. When the memory is above the target usage, we are drastic in increasing the threshold, but when the usage is below the target we are cautious in decreasing it. By measuring the highest and lowest impact the increase of threshold has on the number of flows in the flow memory, we arrived to a value of 3 for *adjustup*, 1 for *adjustdown* in the case of sample and hold and 0.5 for multistage filters.

6.1 Dimensioning the multistage filter

Even if we have the correct constants for the threshold adaptation algorithm, there are other configuration parameters for the multistage filter we need to set. Our aim in this section is not to derive the exact optimal values for the configuration parameters of the multistage filters. Due to the dynamic threshold adaptation, the device will work even if we use suboptimal values for the configuration parameters. Nevertheless we want to avoid using configuration parameters that would lead the dynamic adaptation to stabilize at a value of the threshold that is significantly higher than the one for the optimal configuration.

We assume that design constraints limit the total amount of memory we can use for the stage counters and the flow memory, but we have no restrictions on how to divide it between the filter and the flow memory. Since the number of per packet memory accesses might be limited, we assume that we might have a limit on the number of stages. We want to see how we should divide the available memory between the filter and the flow memory and how many stages to use. We base our configuration parameters on some knowledge of the traffic mix (the number of active flows and the percentage of large flows that are long lived).

We first introduce a simplified model of how the multistage filter works. Measurements confirm this model is closer to the actual behavior of the filters than the conservative analysis. Because of shielding the old large flows do not affect the filter. We assume that because of conservative update only the counters to which the new large flows hash reach the threshold. Let l be the number of large flows and Δl be the number of new large flows. We approximate the probability of a small flow passing one stage by $\Delta l/b$ and of passing the whole filter by $(\Delta l/b)^d$. This gives us the number of false positives in each interval $fp = n(\Delta l/b)^d$. The number of memory locations used at the end of a measurement interval consists of the large flows and the false positives of the previous interval and the new large flows and the new false positives $m = l + \Delta l + 2 * fp$. To be able to establish a tradeoff between using the available memory for the filter or the flow memory, we need to know the relative cost of a counter and a flow entry. Let r denote the ratio between the size of a counter and the size of an entry. The amount of memory used by the filter is going to be equivalent to $b * d * r$ entries. To determine the optimal number of counters per stage given a certain number of large flows, new large flows and stages, we take the derivative of the total memory with respect to b. Equation 4 gives the optimal value for b and Equation 5 gives the total amount of memory required with this choice of b.

$$b = \Delta l \sqrt[d+1]{\frac{2n}{r\Delta l}} \tag{4}$$

$$m_{total} = l + \Delta l + (d+1)r\Delta l \sqrt[d+1]{\frac{2n}{r\Delta l}} \tag{5}$$

We make a further simplifying assumption that the ratio between Δl and l (related to the flow arrival rate) doesn't depend on the threshold. Measurements confirm that this is a good approximation for wide ranges of the threshold. For the MAG trace, when we define the flows at the granularity of TCP connections $\Delta l/l$ is around 44%, when defining flows based on destination IP 37% and when defining them as AS pairs 19%. Let M be the number of entries the available memory can hold. We solve Equation 5 with respect to l for all possible values of d from 2 to the limit on the number of memory accesses we can afford per packet. We choose the depth of the filter that gives the largest l and compute b based on that value.

7. MEASUREMENTS

In Section 4 and Section 5 we used *theoretical* analysis to understand the effectiveness of our algorithms. In this section, we turn to *experimental* analysis to show that our algorithms behave much better on real traces than the (reasonably good) bounds provided by the earlier theoretical analysis and compare them with Sampled NetFlow.

We start by describing the traces we use and some of the configuration details common to all our experiments. In Section 7.1.1 we compare the measured performance of the sample and hold algorithm with the predictions of the analytical evaluation, and also evaluate how much the various improvements to the basic algorithm help. In Section 7.1.2 we evaluate the multistage filter and the improvements that apply to it. We conclude with Section 7.2 where we compare complete traffic measurement devices using our two algorithms with Cisco's Sampled NetFlow.

We use 3 unidirectional traces of Internet traffic: a 4515 second "clear" one (MAG+) from CAIDA (captured in August 2001 on an OC-48 backbone link between two ISPs) and two 90 second anonymized traces from the MOAT project of NLANR (captured in September 2001 at the access points to the Internet of two large universities on an OC-12 (IND) and an OC-3 (COS)). For some of the experiments use only the first 90 seconds of trace MAG+ as trace MAG.

In our experiments we use 3 different definitions for flows. The first definition is at the granularity of TCP connections: flows are defined by the 5-tuple of source and destination IP address and port and the protocol number. This definition is close to that of Cisco NetFlow. The second definition uses the destination IP address as a flow identifier. This is a definition one could use to identify at a router ongoing (distributed) denial of service attacks. The third definition uses the source and destination autonomous system as the flow identifier. This is close to what one would use to determine traffic patterns in the network. We cannot use this definition with the anonymized traces (IND and COS) because we cannot perform route lookups on them.

Table III describes the traces we used. The number of active flows is given for all applicable flow definitions. The reported values are the smallest, largest and average value over the measurement intervals of the respective traces. The number of megabytes per interval is also given as the smallest, average and largest value. Our traces use only between 13% and 27% of their respective link capacities.

The best value for the size of the measurement interval depends both on the application and the traffic mix. We chose to use a measurement interval of 5 seconds in all our experiments. [Estan and Varghese 2002] gives the measurements we base this decision on. Here we only note that in all cases 99% or more of the

Trace	Number of flows (min/avg/max)			MB/interval
	5-tuple	destination IP	AS pair	(min/max)
MAG+	93,437/98,424/105,814	40,796/42,915/45,299	7,177/7,401/7,775	201.0/284.2
MAG	99,264/100,105/101,038	43,172/43,575/43,987	7,353/7,408/7,477	255.8/273.5
IND	13,746/14,349/14,936	8,723/8,933/9,081	-	91.37/99.70
COS	5,157/5,497/5,784	1,124/1,146/1,169	-	14.28/18.70

Table III. The traces used for our measurements

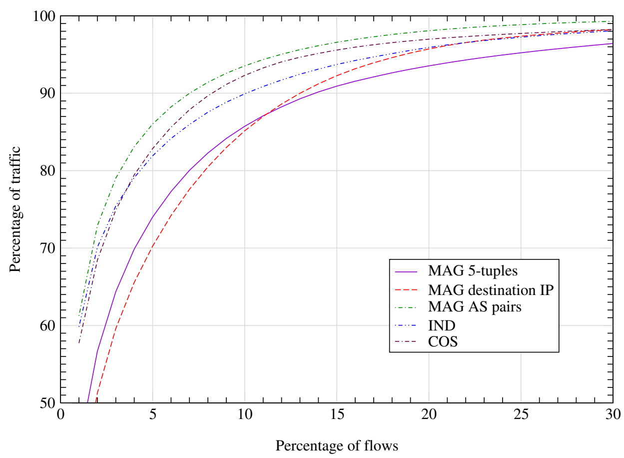

Fig. 7. Cumulative distribution of flow sizes for various traces and flow definitions

packets (weighted by packet size) arrive within 5 seconds of the previous packet belonging to the same flow.

Since our algorithms are based on the assumption that a few heavy flows dominate the traffic mix, we find it useful to see to what extent this is true for our traces. Figure 7 presents the cumulative distributions of flow sizes for the traces MAG, IND and COS for flows defined by 5-tuples. For the trace MAG we also plot the distribution for the case where flows are defined based on destination IP address, and for the case where flows are defined based on the source and destination ASes. As we can see, the top 10% of the flows represent between 85.1% and 93.5% of the total traffic validating our original assumption that a few flows dominate.

7.1 Comparing Theory and Practice

Here we summarize our most important results that compare the theoretical bounds with the results on actual traces, and quantify the benefits of various optimizations. In Appendix B we discuss more measurement results for sample and hold and in Appendix C more results for multistage filters.

Algorithm	Maximum memory usage (entries)/ Average error				
	MAG 5tuple	MAG dstIP	MAG ASpair	IND 5tuple	COS 5tuple
General bound	16,385/25%	16,385/25%	16,385/25%	16,385/25%	16,385/25%
Zipf bound	8,148/25%	7,441/25%	5,489/25%	6,303/25%	5,081/25%
Sample and hold	2,303/24.3%	1,964/24.1%	714/24.40%	1,313/23.8%	710/22.17%
+pres. entries	3,832/4.67%	3,213/3.28%	1,038/1.32%	1,894/3.04%	1,017/6.6%
+early removal	2,659/3.89%	2,294/3.16%	803/1.18%	1,525/2.92%	859/5.46%

Table IV. Summary of sample and hold measurements for a threshold of 0.025% and an oversampling of 4

7.1.1 *Summary of findings about sample and hold.* Table IV summarizes our results for a single configuration: a threshold of 0.025% of the link with an oversampling of 4. We ran 50 experiments (with different random hash functions) on each of the reported traces with the respective flow definitions. The table gives the maximum memory usage over the 900 measurement intervals and the ratio between average error for large flows and the threshold.

The first row presents the *theoretical* bounds that hold without making any assumption about the distribution of flow sizes and the number of flows. These are not the bounds on the expected number of entries used (which would be 16,000 in this case), but high probability bounds.

The second row presents *theoretical* bounds assuming that we know the number of flows and know that their sizes have a *Zipf distribution* with a parameter of $\alpha = 1$. Note that the relative errors predicted by theory may appear large (25%) but these are computed for a very low threshold of 0.025% and only apply to flows exactly at the threshold.[16]

The third row shows the actual values we measured for the basic sample and hold algorithm. The actual memory usage is much below the bounds. The first reason is that the links are lightly loaded and the second reason (partially captured by the analysis that assumes a Zipf distribution of flows sizes) is that large flows have many of their packets sampled. The average error is very close to its expected value.

The fourth row presents the effects of preserving entries. While this increases memory usage (especially where large flows do not have a big share of the traffic) it significantly reduces the error for the estimates of the large flows, because there is no error for large flows identified in previous intervals. This improvement is most noticeable when we have many long lived flows.

The last row of the table reports the results when preserving entries as well as using an early removal threshold of 15% of the threshold (see Appendix B for why this is a good value). We compensated for the increase in the probability of false negatives early removal causes by increasing the oversampling to 4.7. The average error decreases slightly. The memory usage decreases, especially in the cases where

[16]We defined the relative error by dividing the average error by the size of the threshold. We could have defined it by taking the average of the ratio of a flow's error to its size but this makes it difficult to compare results from different traces.

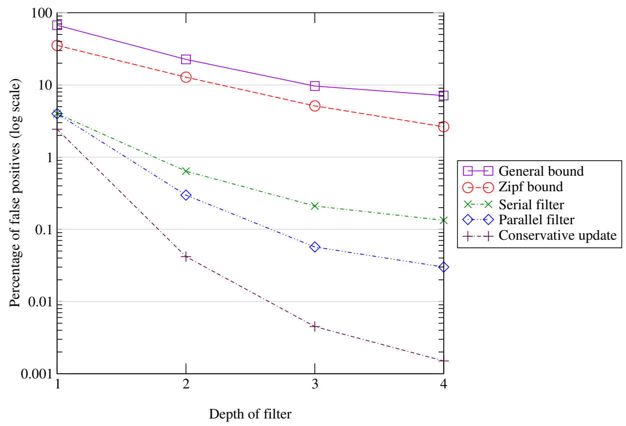

Fig. 8. Filter performance for a stage strength of k=3

preserving entries caused it to increase most.

We performed measurements on many more configurations. The results are in general similar to the ones from Table IV, so we only emphasize some noteworthy differences. First, when the expected error approaches the size of a packet, we see significant decreases in the average error. Our analysis assumes that we sample at the byte level. In practice, if a certain packet gets sampled all its bytes are counted, including the ones before the byte that was sampled.

Second, preserving entries reduces the average error by 70% - 95% and increases memory usage by 40% - 70%. These figures do not vary much as we change the threshold or the oversampling. Third, an early removal threshold of 15% reduces the memory usage by 20% - 30%. The size of the improvement depends on the trace and flow definition and it increases slightly with the oversampling.

7.1.2 *Summary of findings about multistage filters.* Figure 8 summarizes our findings about configurations with a stage strength of $k = 3$ for our most challenging trace: MAG with flows defined at the granularity of TCP connections. It represents the percentage of small flows (log scale) that passed the filter for depths from 1 to 4 stages. We used a threshold of a 4096th of the maximum traffic. The first (i.e., topmost and solid) line represents the bound of Theorem 3. The second line below represents the improvement in the theoretical bound when we assume a Zipf distribution of flow sizes. Unlike in the case of sample and hold we used the maximum traffic, not the link capacity for computing the theoretical bounds. This results in much tighter theoretical bounds.

The third line represents the measured average percentage of false positives of a serial filter, while the fourth line represents a parallel filter. We can see that both are at least 10 times better than the stronger of the theoretical bounds. As the number of stages goes up, the parallel filter gets better than the serial filter by up to a factor of 4. The last line represents a parallel filter with conservative update which gets progressively better than the parallel filter by up to a factor of

20 as the number of stages increases. We can see that all lines are roughly straight; this indicates that the percentage of false positives decreases exponentially with the number of stages.

Measurements on other traces show similar results. The difference between the bounds and measured performance is even larger for the traces where the largest flows are responsible for a large share of the traffic. With conservative update and without preserving entries the average error is very close to the threshold. Preserving entries reduces the average error in the estimates by 70% to 85% because the estimates for long lived flows are exact in all but the first measurement interval. The improvements in the results due to preserving entries depend on the traffic mix. Preserving entries increases the number of flow memory entries used by up to 30%. By effectively increasing stage strength k, shielding considerably strengthens weak filters. This can lead to reducing the number of entries by as much as 70%.

7.2 Evaluation of complete traffic measurement devices

We now present our final comparison between sample and hold, multistage filters and sampled NetFlow. We perform the evaluation on our long OC-48 trace, MAG+. We assume that our devices can use 1 Mbit of memory (4096 entries[17]) which is well within the possibilities of today's chips. Sampled NetFlow is given unlimited memory and uses a sampling of 1 in 16 packets. We run each algorithms 16 times on the trace with different sampling or hash functions.

Both our algorithms use the adaptive threshold approach. To avoid the effect of initial misconfiguration, we ignore the first 10 intervals to give the devices time to reach a relatively stable value for the threshold. We impose a limit of 4 stages for the multistage filters. Based on heuristics presented in Section 6.1, we use 3114 counters[18] for each stage and 2539 entries of flow memory when using a flow definition at the granularity of TCP connections, 2646 counters and 2773 entries when using the destination IP as flow identifier and 1502 counters and 3345 entries when using the source and destination AS. Multistage filters use shielding and conservative update. Sample and hold uses an oversampling of 4 and an early removal threshold of 15%.

Our purpose is to see how accurately the algorithms measure the largest flows, but there is no implicit definition of what large flows are. We look separately at how well the devices perform for three reference groups: very large flows (above one thousandth of the link capacity), large flows (between one thousandth and a tenth of a thousandth) and medium flows (between a tenth of a thousandth and a hundredth of a thousandth – 15,552 bytes).

For each of these groups we look at two measures of accuracy that we average over all runs and measurement intervals: the percentage of flows not identified and the relative average error. We compute the relative average error by dividing the sum of the absolute values of all errors by the sum of the sizes of all flows. We use the modulus so that positive and negative errors don't cancel out for NetFlow.

[17]Cisco NetFlow uses 64 bytes per entry in cheap DRAM. We conservatively assume that the size of a flow memory entry will be 32 bytes (even though 16 or 24 are also plausible).

[18]We conservatively assume that we use 4 bytes for a counter even though 3 bytes would be enough.

Group (flow size)	Unidentified flows / Average error		
	Sample and hold	Multistage filters	Sampled NetFlow
> 0.1%	0% / 0.07508%	0% / 0.03745%	0% / 9.020%
0.1 . . . 0.01%	1.797% / 7.086%	0% / 1.090%	0.02132% / 22.02%
0.01 . . . 0.001%	77.01% / 61.20%	54.70% / 43.87%	17.72% / 50.27%

Table V. Comparison of traffic measurement devices with flow IDs defined by 5-tuple

Group (flow size)	Unidentified flows / Average error		
	Sample and hold	Multistage filters	Sampled NetFlow
> 0.1%	0% / 0.02508%	0% / 0.01430%	0% / 5.720%
0.1 . . . 0.01%	0.4289% / 3.153%	0% / 0.9488%	0.01381% / 20.77%
0.01 . . . 0.001%	65.72% / 51.19%	49.91% / 39.91%	11.54% / 46.59%

Table VI. Comparison of traffic measurement devices with flow IDs defined by destination IP

For the unidentified flows, we consider that the error is equal to their total traffic. Tables V to VII present the results for the 3 different flow definitions.

When using the source and destination AS as flow identifier, the situation is different from the other two cases because the average number of active flows (7,401) is not much larger than the number of memory locations that we can accommodate in our SRAM (4,096), so we will discuss this case separately. In the first two cases, we can see that both our algorithms are much more accurate than sampled NetFlow for large and very large flows. For medium flows the average error is roughly the same, but our algorithms miss more of them than sampled NetFlow. Since sample and hold stabilized at thresholds slightly above 0.01% and multistage filters around 0.002% it is normal that so many of the flows from the third group are not detected.

We believe these results (and similar results not presented here) confirm that our algorithms are better than sampled NetFlow at measuring large flows. Multistage filters are always slightly better than sample and hold despite the fact that we have to sacrifice part of the memory for stage counters. However, tighter algorithms for threshold adaptation can possibly improve both algorithms.

In the third case since the average number of very large, large and medium flows (1,107) was much below the number of available memory locations and these flows were mostly long lived, both of our algorithms measured all these flows very accurately. Thus, even when the number of flows is only a few times larger than the number of active flows, our algorithms ensure that the available memory is used to accurately measure the largest of the flows and provide graceful degradation in case that the traffic deviates very much from the expected (e.g. more flows).

8. IMPLEMENTATION ISSUES

We briefly describe implementation issues. Sample and Hold is easy to implement even in a network processor because it adds only one memory reference to packet processing, assuming sufficient SRAM for flow memory and assuming an associative memory. For small flow memory sizes, adding a CAM is quite feasible. Alterna-

Group (flow size)	Unidentified flows / Average error		
	Sample and hold	Multistage filters	Sampled NetFlow
$> 0.1\%$	0% / 0.000008%	0% / 0.000007%	0% / 4.877%
$0.1 \ldots 0.01\%$	0% / 0.001528%	0% / 0.001403%	0.002005% / 15.28%
$0.01 \ldots 0.001\%$	0.000016% / 0.1647%	0% / 0.1444%	5.717% / 39.87%

Table VII. Comparison of traffic measurement devices with flow IDs defined by the source and destination AS

tively, one can implement an associative memory using a hash table and storing all flow IDs that collide in a much smaller CAM.

Multistage filters are harder to implement using a network processor because they need multiple stage memory references. However, multistage filters are easy to implement in an ASIC as the following feasibility study shows. [Huber 2001] describes a chip designed to implement a parallel multistage filter with 4 stages of 4K counters each and a flow memory of 3584 entries. The chip runs at OC-192 line speeds. The core logic consists of roughly 450,000 transistors that fit on 2mm x 2mm on a .18 micron process. Including memories and overhead, the total size of the chip would be 5.5mm x 5.5mm and would use a total power of less than 1 watt, which put the chip at the low end of today's IC designs.

9. CONCLUSIONS

Motivated by measurements that show that traffic is dominated by a few heavy hitters, our paper tackles the problem of directly identifying the heavy hitters without keeping track of potentially millions of small flows. Fundamentally, Table I shows that our algorithms have a much better scaling of estimate error (inversely proportional to memory size) than provided by the state of the art Sampled NetFlow solution (inversely proportional to the *square root* of the memory size). On actual measurements, our algorithms with optimizations do several orders of magnitude better than predicted by theory.

However, comparing Sampled NetFlow with our algorithms is more difficult than indicated by Table I. This is because Sampled NetFlow does not process every packet and hence can afford to use large DRAM. Despite this, results in Table II and in Section 7.2 show that our algorithms are much more accurate for small intervals than NetFlow. In addition, unlike NetFlow, our algorithms provide exact values for long-lived large flows, provide provable lower bounds on traffic that can be reliably used for billing, avoid resource-intensive collection of large NetFlow logs, and identify large flows very fast.

The above comparison only indicates that the algorithms in this paper may be better than using Sampled NetFlow when the only problem is that of identifying heavy hitters, and when the manager has a precise idea of which flow definitions are interesting. But NetFlow records allow managers to mine *a posteriori* patterns in data they did not anticipate, while our algorithms rely on efficiently identifying stylized patterns that are defined *a priori*. To see why this may be insufficient, imagine that CNN suddenly gets flooded with web traffic. How could a manager realize before the event that the interesting flow definition to watch for is a multipoint-to-point flow, defined by destination address and port numbers?

The last example motivates an interesting open question. Is it possible to generalize the algorithms in this paper to automatically extract flow definitions corresponding to large flows? A second open question is to deepen our theoretical analysis to account for the large discrepancies between theory and experiment.

We end by noting that measurement problems (data volume, high speeds) in networking are similar to the measurement problems faced by other areas such as data mining, architecture, and even compilers. For example, Sastry et al. recently proposed using a Sampled NetFlow-like strategy to obtain dynamic instruction profiles in a processor for later optimization[Sastry et al. 2001]. Narayanasamy et al. show[Narayanasamy et al. 2003] that multistage filters with conservative update can improve the results of [Sastry et al. 2001]. Thus the techniques in this paper may be of utility to other areas, and the techniques in these other areas may of utility to us.

10. ACKNOWLEDGEMENTS

We thank K. Claffy, D. Moore, F. Baboescu and the anonymous reviewers for valuable comments. This work was made possible by a grant from NIST for the Sensilla Project, and by NSF Grant ANI 0074004.

REFERENCES

Altman, J. and Chu, K. 2001. A proposal for a flexible service plan that is attractive to users and internet service providers. In *IEEE Proceedings of the INFOCOM*.

Bloom, B. 1970. Space/time trade-offs in hash coding with allowable errors. In *Commun. ACM*. Vol. 13. 422–426.

Brownlee, N., Mills, C., and Ruth, G. 1999. Traffic flow measurement: Architecture. RFC 2722.

Burrows, M., Erlingson, U., Leung, S.-T., Vandevoorde, M., Waldspurger, C. A., and Weihl, K. W. W. 2000. Efficient and flexible value sampling. In *ASPLOS*.

Cohen, S. and Matias, Y. 2003. Spectral bloom filters. In *SIGMOD/PODS*.

Duffield, N., Lund, C., and Thorup, M. 2001. Charging from sampled network usage. In *SIGCOMM Internet Measurement Workshop*.

Duffield, N. G. and Grossglauser, M. 2000. Trajectory sampling for direct traffic observation. In *Proceedings of the ACM SIGCOMM*. 271–282.

Estan, C. and Varghese, G. 2002. New directions in traffic measurement and accounting. Tech. Rep. 0699, UCSD CSE Department. Feb.

Fang, M., Shivakumar, N., Garcia-Molina, H., Motwani, R., and Ullman, J. D. 1998. Computing iceberg queries efficiently. In *International Conference on Very Large Data Bases*. 307–317.

Fang, W. and Peterson, L. 1999. Inter-as traffic patterns and their implications. In *Proceedings of IEEE GLOBECOM*.

Feldmann, A., Greenberg, A., Lund, C., Reingold, N., Rexford, J., and True, F. 2000. Deriving traffic demands for operational ip networks: Methodology and experience. In *Proceedings of the ACM SIGCOMM*. 257–270.

Feng, W.-c., Kandlur, D. D., Saha, D., and Shin, K. G. 2001. Stochastic fair blue: A queue management algorithm for enforcing fairness. In *IEEE Proceedings of the INFOCOM*.

Gibbons, P. B. and Matias, Y. 1998. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the ACM SIGMOD*. 331–342.

Huber, J. 2001. Design of an oc-192 flow monitoring chip. Class Project.

Karp, R. M., Papadimitriou, C. H., and Shenker, S. 2003. A simple algorithm for finding frequent elements in streams and bags.

Mackie-Masson, J. and Varian, H. 1995. *Public Access to the Internet.* MIT Press, Chapter Pricing the Internet.

Mahajan, R., Bellovin, S. M., Floyd, S., Ioannidis, J., Paxson, V., and Shenker, S. 2001. Controlling high bandwidth aggregates in the network. http://www.aciri.org/pushback/.

Moore, D. 2001. Personal conversation. also see caida analysis of code-red. http://www.caida.org/ analysis/ security/ code-red/.

Narayanasamy, S., Sherwood, T., Sair, S., Calder, B., and Varghese, G. 2003. Catching accurate profiles in hardware. In *HPCA*.

NetFlow. Cisco netflow. http://www.cisco.com /warp /public /732 /Tech /netflow.

Pan, R., Breslau, L., Prabhakar, B., and Shenker, S. 2001. Approximate fairness through differential dropping. Tech. rep., ACIRI.

Patterson, D. A. and Hennessy, J. L. 1998. *Computer Organization and Design*, second ed. Morgan Kaufmann, 619.

Sastry, S., Bodik, R., and Smith, J. E. 2001. Rapid profiling via stratified sampling. In *28th. International Symposium on Computer Architecture*. 278–289.

Shaikh, A., Rexford, J., and Shin, K. G. 1999. Load-sensitive routing of long-lived ip flows. In *Proceedings of the ACM SIGCOMM*.

Shenker, S., Clark, D., Estrin, D., and Herzog, S. 1996. Pricing in computer networks: Reshaping the research agenda. In *ACM Computer Communications Review*. Vol. 26. 19–43.

Smitha, Kim, I., and Reddy, A. L. N. 2001. Identifying long term high rate flows at a router. In *Proceedings of High Performance Computing*.

Thomson, K., Miller, G. J., and Wilder, R. 1997. Wide-area traffic patterns and characteristics. In *IEEE Network*.

Tong, D. and Reddy, A. L. N. 1999. Qos enhancement with partial state. In *International Workshop on QOS*.

A. DETAILS OF THE ANALYSIS FOR MULTISTAGE FILTERS

This appendix presents the full derivation for Theorem 3. We use the same notation as in Section 4.2. We first derive the necessary lemmas.

LEMMA 5. *The probability of a flow of size $s \geq 0$ passing one stage of the filter is bound by $p_s \leq \frac{1}{k}\frac{T}{T-s}$. If $s < T\frac{k-1}{k}$ this bound is below 1.*

Proof Let's assume that the flow is the last one to arrive into the bucket. This does not increase its chance to pass the stage, on the contrary: in reality it might have happened that all packets belonging to the flow arrived before the bucket reached the threshold and the flow was not detected even if the bucket went above the threshold in the end. Therefore the probability of the flow passing the stage is not larger than the probability that the bucket it hashed to reaches T. The bucket of the flow can reach T only if the other flows hashing into the bucket add up to at least $T-s$. The total amount of traffic belonging to other flows is $C-s$. Therefore, the maximum number of buckets in which the traffic of other flows can reach at least $T-s$ is $\lfloor\frac{C-s}{T-s}\rfloor$. The probability of a flow passing the filter is bound by the probability of it being hashed into such a bucket.

$$p_s \leq \frac{\lfloor\frac{C-s}{T-s}\rfloor}{b} \leq \frac{C}{b(T-s)} = \frac{1}{k}\frac{T}{T-s} \blacksquare$$

Based on this lemma we can compute the probability that a small flow passes the parallel multistage filter.

LEMMA 1. *Assuming the hash functions used by different stages are independent, the probability of a flow of size s passing a parallel multistage filter is bound by $p_s \leq \left(\frac{1}{k}\frac{T}{T-s}\right)^d$.*

Proof A flow passes the filter only if it passes all the stages. Since all stages are updated in the same way for the parallel filter, lemma 5 applies to all of them. Since the hash functions are independent, the probability of the flow passing all of the stages equals the product of the probabilities for every stage. $\blacksquare$

Now we can give the bound on the number of flows passing a multistage filter.

THEOREM 3. *The expected number of flows passing a parallel multistage filter is bound by*

$$E[n_{pass}] \leq max\left(\frac{b}{k-1}, n\left(\frac{n}{kn-b}\right)^d\right) + n\left(\frac{n}{kn-b}\right)^d \tag{6}$$

Proof Let s_i be the sequence of flow sizes present in the traffic mix. Let n_i the number of flows of size s_i. $h_i = \frac{n_i\ s_i}{C}$ is the share of the total traffic the flows of size s_i are responsible for. It is immediate that $\sum n_i = n$, and $\sum h_i = 1$. By lemma 1 the expected number of flows of size s_i to pass the filter is $E[n_{i_{pass}}] = n_i p_{s_i} \leq n_i\ min(1, (\frac{1}{k}\frac{T}{T-s_i})^d)$. By the linearity of expectation we have $E[n_{pass}] = \sum E[n_{i_{pass}}]$.

To be able to bound $E[n_{pass}]$, we will divide flows in 3 groups by size. The largest flows are the ones we cannot bound p_{s_i} for. These are the ones with $s_i > T\frac{k-1}{k}$. The smallest flows are the ones below the average flow size of $\frac{C}{n}$. For these $p_{s_i} \leq p_{\frac{C}{n}}$.

The number of below average flows is bound by n. For all these flows taken together $E[n_{small_{pass}}] \leq np_{\frac{C}{n}}$. The middle group is that of flows between $\frac{C}{n}$ and $T\frac{k-1}{k}$.

$$\begin{aligned}
E[n_{pass}] &= \sum E[n_{i_{pass}}] = \sum_{s_i > T\frac{k-1}{k}} E[n_{i_{pass}}] + \sum_{\frac{C}{n} \leq s_i \leq T\frac{k-1}{k}} E[n_{i_{pass}}] + \sum_{s_i < \frac{C}{n}} E[n_{i_{pass}}] \\
&\leq \sum_{s_i > T\frac{k-1}{k}} \frac{h_i C}{s_i} + \sum_{\frac{C}{n} \leq s_i \leq T\frac{k-1}{k}} \frac{h_i C}{s_i} \left(\frac{1}{k}\frac{T}{T-s_i}\right)^d + n\left(\frac{1}{k}\frac{T}{T-\frac{C}{n}}\right)^d \\
&\leq C\left(\sum_{s_i > T\frac{k-1}{k}} h_i \frac{1}{T\frac{k-1}{k}} + \sum_{\frac{C}{n} \leq s_i \leq T\frac{k-1}{k}} h_i \frac{1}{s_i}\left(\frac{1}{k}\frac{T}{T-s_i}\right)^d\right) + n\left(\frac{1}{k}\frac{T}{T-\frac{C}{n}}\right)^d
\end{aligned}$$

It's maybe easier to follow how the proof proceeds from here on if we assume that we have an adversary that tries to arrange the flows on purpose so that the largest number of possible flows passes the filter. But this adversary has a budget limited by the total amount of traffic it can send (the h_is have to add up to (at most) one because he cannot send more than the link bandwidth). We can see that the adversary can achieve the highest number by spending the traffic it allocates to flows above $T\frac{k-1}{k}$ to flows exactly at $T\frac{k-1}{k}$. This is equivalent to noticing that substituting all flows from this group with a number of flows of size $T\frac{k-1}{k}$ that generate the same amount of traffic is guaranteed to not decrease the lower bound for $E[n_{pass}]$. The next step is based on the observation that the number of flows passing the filter is maximized when the adversary chooses the size of flows in the middle group that maximizes the number of flows expected to pass the filter for a given amount of total traffic.

$$\begin{aligned}
E[n_{pass}] &\leq C\left(\sum_{\frac{C}{n} \leq s_i \leq T\frac{k-1}{k}} h_i \frac{1}{s_i}\left(\frac{1}{k}\frac{T}{T-s_i}\right)^d\right) + n\left(\frac{1}{k}\frac{T}{T-\frac{C}{n}}\right)^d \\
&\leq C \max_{\frac{C}{n} \leq s_i \leq T\frac{k-1}{k}} \frac{1}{s_i}\left(\frac{1}{k}\frac{T}{T-s_i}\right)^d + n\left(\frac{1}{k}\frac{T}{T-\frac{C}{n}}\right)^d
\end{aligned}$$

Next we determine the maximum of the function $f(x) = \frac{1}{x}(\frac{1}{T-x})^d$ on the domain $[\frac{C}{n}, T\frac{k-1}{k}]$.

$$f'(x) = -\frac{1}{x^2}\left(\frac{1}{T-x}\right)^d + \frac{1}{x}\frac{d}{(T-x)^{d+1}} = \frac{1}{x}\frac{1}{(T-x)^d}\left(-\frac{1}{x} + \frac{d}{T-x}\right)$$

Within $[\frac{C}{n}, T\frac{k-1}{k}]$ $f'(x) = 0$ for $x = \frac{T}{d+1}$ (if it is in the interval), $f'(x) < 0$ to the left of this value and $f'(x) > 0$ to the right of it. Therefore this represents a minimum for $f(x)$. Therefore the maximum of $f(x)$ will be obtained at one of the ends of the interval $C(\frac{T}{k})^d f(T\frac{k-1}{k}) = \frac{C}{T\frac{k-1}{k}} = \frac{b}{k-1}$ or $C(\frac{T}{k})^d f(\frac{C}{n}) = n(\frac{1}{k}\frac{T}{T-\frac{C}{n}})^d = n(\frac{nT}{knT-kC})^d = n(\frac{n}{kn-b})^d$. Substituting these values we obtain the bound. ■

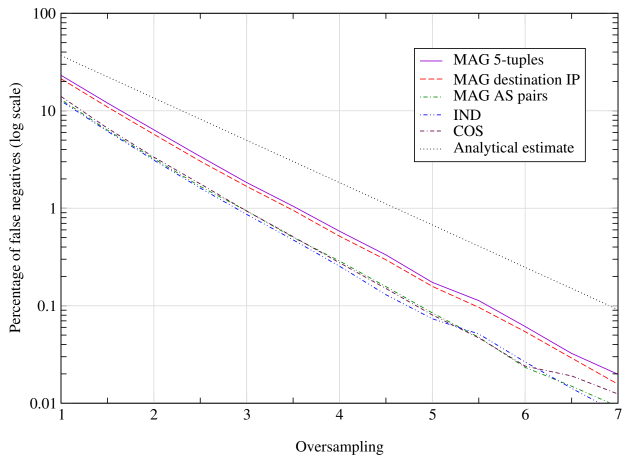

Fig. 9. Percentage of false negatives as the oversampling changes

B. MEASUREMENTS OF SAMPLE AND HOLD

In this appendix we present a detailed discussion of our measurements of the performance on sample and hold and its optimizations. Some of the less important results were omitted (all results are in [Estan and Varghese 2002]), but all results are discussed. We first compare the measured performance of the sample and hold algorithm to the values predicted by our analysis. Next we measure the improvement introduced by preserving entries across measurement intervals. In the last subsection we measure the effect of early removal and determine a good value for the early removal threshold.

We use 3 measures for the performance of the sample and hold algorithm: the average percentage of large flows that were not identified (false negatives), the average error of the traffic estimates for the large flows and the maximum number of locations used in the flow memory.

B.1 Comparing the behavior of the basic algorithm to the analytic results

Our first set of experiments looks at the effect of oversampling on the performance of sample and hold. We configure sample and hold to measure the flows above 0.01% of the link bandwidth and vary the oversampling factor from 1 to 7 (a probability of between 37% and less than 0.1% of missing a flow at the threshold). We perform each experiment for the trace MAG, IND and COS and for the trace MAG we use all 3 flow definitions. For each configuration, we perform 50 runs with different random functions for choosing the sampled packets. Figure 9 shows the percentage of false negatives (the Y axis is logarithmic). We also plot the probability of false negatives predicted by our conservative analysis. The measurement results are considerably better than predicted by the analysis. The reason is that the analysis assumes that the size of the large flow is exactly equal to the threshold while most of the large flows are much above the threshold making them much more likely to be identified.

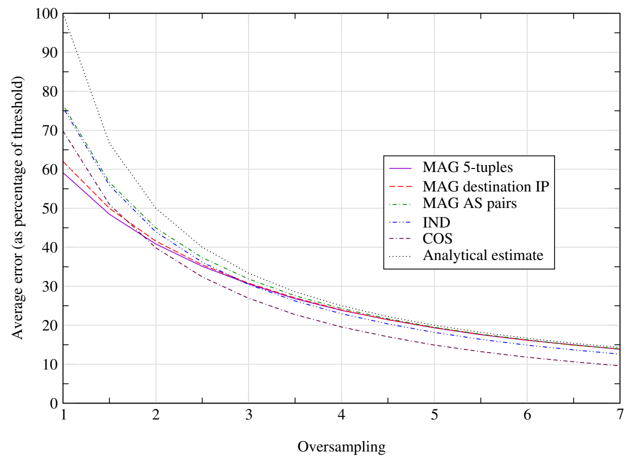

Fig. 10. Average error in the traffic estimates for large flows

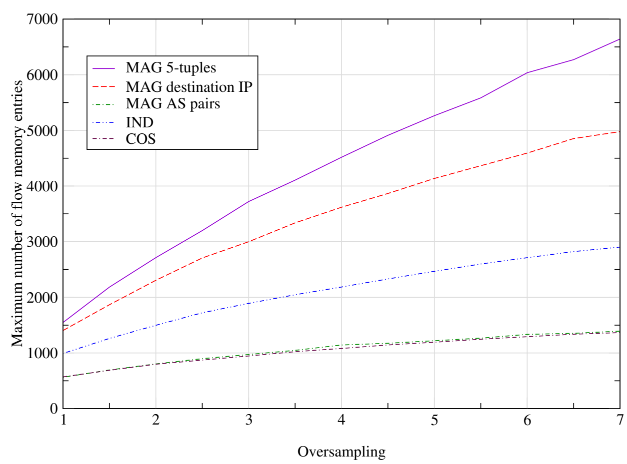

Fig. 11. Maximum number of flow memory entries used

The configurations with many large flows close to the threshold have false negative ratios closest to the results of our conservative analysis. The results confirm that the probability of false negatives decreases exponentially as the oversampling increases. Figure 10 shows the average error in the estimate of the size of an identified large flow. The measured error is slightly below the error predicted by the analysis. The results confirm that the average error of the estimates is proportional to the inverse of the oversampling. Figure 11 shows the maximum over the 900 measurement intervals for the number of entries of flow memory used. The measurement results

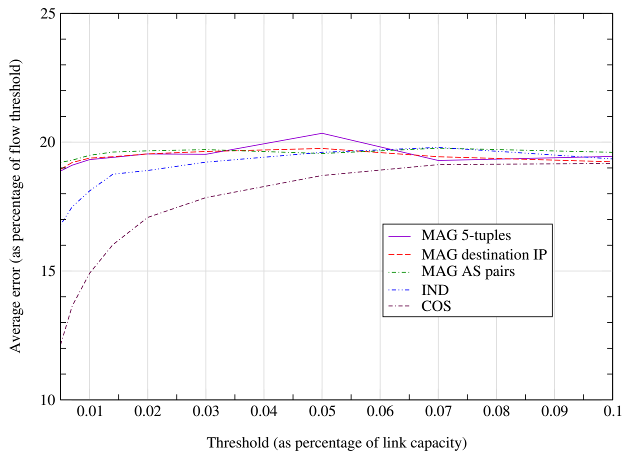

Fig. 12. Average error in the traffic estimates for large flows

are more than an order of magnitude lower than the bound from Section 4.1.2. There are two main reasons: the links are lightly loaded (between 13% and 27%) and many of the sampled packets belonging to large flows do not create new entries in the flow memory. The results also show that the number of entries used depends on the number of active flows and the dependence is stronger as the sampling probability (the oversampling) increases.

The next set of experiments looks at how the choice of the threshold influences the performance of the sample and hold algorithm. We run the algorithm with a fixed oversampling factor of 5 for thresholds between 0.005% and 0.1% of the link bandwidth. The most interesting result is Figure 12 showing the average error in the estimate of the size of an identified large flow. As expected, the actual values are usually slightly below the expected error of 20% of the threshold. The only significant deviations are for the traces IND and especially COS at very small values of the threshold. The explanation is that the threshold approaches the size of a large packet (e.g. a threshold of 0.005% on an OC3 (COS) corresponds to 4860 bytes while the size of most packets of the large flows is 1500 bytes). Our analysis assumes that we sample at the byte level. In practice, if a certain packet gets sampled all its bytes are counted, including the ones before the byte that was sampled.

B.2 The effect of preserving entries

For all traces, we performed two sets of experiments: with fixed threshold and varying oversampling and with fixed oversampling and varying the threshold. The improvement introduced by preserving entries is not influenced much by the oversampling but it is influenced considerably by the choice of the threshold. We conjecture that this happens because the magnitude of the improvement depends on the distribution of the durations for large flows and this changes as we change the threshold because the mix of large flows changes. Preserving entries reduces

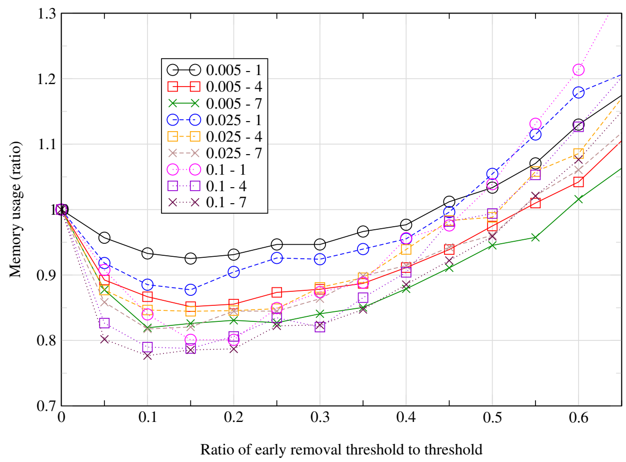

Fig. 13. Effect of early removal on memory usage

Trace+flow definition	False neg. change min/median/max	Average error change min/median/max	Memory change min/median/max
MAG 5-tuple	0%/95.2%/200%	77.4%/90.6%/92.6%	64.5%/69.3%/81.0%
MAG destination IP	0%/90.5%/100%	79.9%/90.4%/98.2%	66.0%/72.3%/87.3%
MAG AS pairs	50%/92.4%/100%	78.7%/88.9%/93.2%	74.8%/80.5%/91.8%
IND 5-tuple	55.6%/92.0%/160%	81.4%/89.5%/96.2%	73.6%/80.5%/91.4%
COS 5-tuple	0%/84.5%/104%	77.5%/85.0%/92.3%	78.6%/82.6%/92.5%

Table VIII. Various measures of performance when using an early removal threshold of 15% of the threshold as compared with the values without early removal

the probability of false negatives by 50% - 85%. It reduces the average error by 70% - 95%. The reduction is strongest when large flows are long lived. Preserving entries increases memory usage by 40% - 70%. The increase is smallest when large flows make up a larger share of the traffic.

B.3 The effect of early removal

To measure the effect of early removal, we used 9 configurations with oversampling of 1, 4 and 7 and with thresholds of 0.005% 0.025% and 0.1% of the link bandwidth. For each of these configurations, we measure a range of values for the early removal threshold. We adjust the oversampling such that the probability of missing a flow at the threshold stays the same as without early removal (see Section 4.1.4 for details). The point of these experiments is to obtain the value for the early removal threshold that results in the smallest possible memory usage. We performed 50 runs on the COS trace for each configuration. The measurements show that the probability of false negatives decreases slightly as the early removal threshold increases. This confirms that we compensate correctly for the large flows that might be removed early (through the increase the oversampling). Results also confirm our expectation

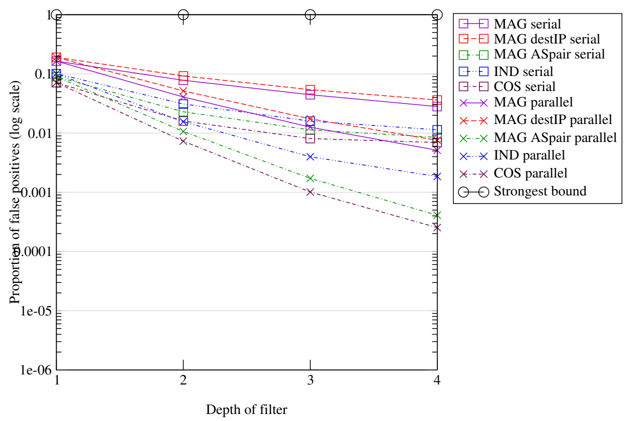

Fig. 14. Actual performance for a stage strength of k=1

that the average error decrease roughly linearly as the early removal threshold increases (due to the compensatory increase in oversampling). Figure 13 shows that there is an optimal value for the early removal threshold (as far as memory usage is concerned) around 15% of the threshold. From the results we can also conclude that the larger the threshold the more memory we save but the less we gain in accuracy with early removal. Also the larger the oversampling, the more we gain in accuracy and memory.

The results for other traces and other flow definitions have very similar trends, but the actual improvements achieved for various metrics are sometimes different. Table VIII has the minimum, median and maximum values (among the 9 configurations) for the 3 metrics of interest when using an early removal threshold of 15% of the threshold. All values are reported as ratios to the values obtained without early removal.

C. MEASUREMENTS OF MULTISTAGE FILTERS

We first compare the performance of serial and parallel multistage filters to the bound of Theorem 3. Next we measure the benefits of conservative update. In the last subsection we measure the combined effect of preserving entries and shielding.

C.1 Comparing the behavior of basic filters to the analytic results

First we compare the number of false positives for serial and parallel filters with the bound of Theorem 3. While the number of flow memory locations used might seem like a more meaningful measure of the performance of the algorithm we use the number of false positives because for strong filters, the number of entries is dominated by the entries of the actual large flows making it harder to distinguish changes of even an order of magnitude in the number of entries occupied by false positives. To make it easier to compare results from different traces and different flow definitions (therefore different numbers of active flows) we actually report the percentage of false positives, not their number. Another important detail is that we

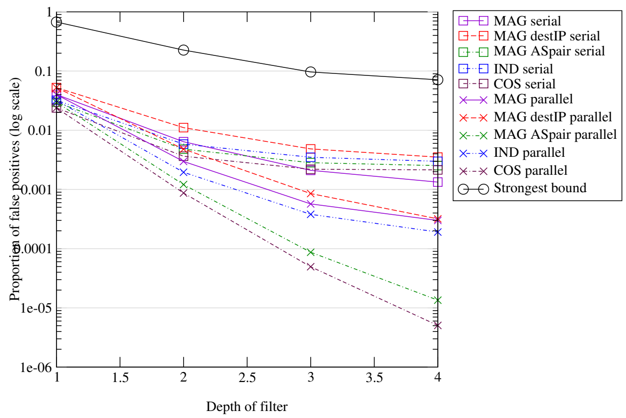

Fig. 15. Actual performance for a stage strength of k=3

express the threshold as a percentage of the maximum traffic, not as a percentage of the link capacity. While actual implementations do not know the traffic in advance, this choice of thresholds gives us information about how the filters would behave under extreme conditions (i.e. a fully loaded link). In this first set of experiments, we fix the threshold to a 4096th of the maximum traffic and vary the stage strength from 1 to 4 and the depth of the filter from 1 to 4 (the number of counters used by the filter is between 4K and 64K). For each configuration we measure 10 runs with different random hash functions. Figure 14 and 15 present the results of our measurements for stage strengths of 1 and 3 (all results are in [Estan and Varghese 2002]). We also represent the strongest bound we obtain from Theorem 3 for the configurations we measure. Note that the y axis is logarithmic.

The results show that the filtering is in general at least an order of magnitude stronger than the bound. Parallel filters are stronger than serial filters with the same configuration. The difference grows from nothing in the degenerate case of a single stage to up to two orders of magnitude for four stages. The actual filtering also depends on the trace and flow definition. We can see that the actual filtering is strongest for the traces and flow definitions for which the large flows strongly dominate the traffic. We can also see that the actual filtering follows the straight lines that denotes exponential improvement with the numbering of stages. For some configurations, after a certain point, the filtering doesn't improve as fast anymore. Our explanation is that the false positives are dominated by a few flows close to threshold. Since the parallel filters clearly outperform the serial ones we use them in all of our subsequent experiments.

C.2 The effect of conservative update

Our next set of experiments evaluates the effect of conservative update. We run experiments with filter depths from 1 to 4. For each configuration we measure 10 runs with different random hash functions. For brevity we only present in figures 16 and 17 the results for stage strengths of 1 and 3. The improvement introduced

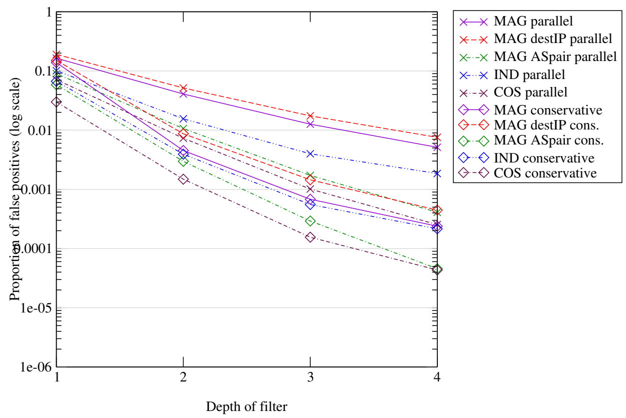

Fig. 16. Conservative update for a stage strength of k=1

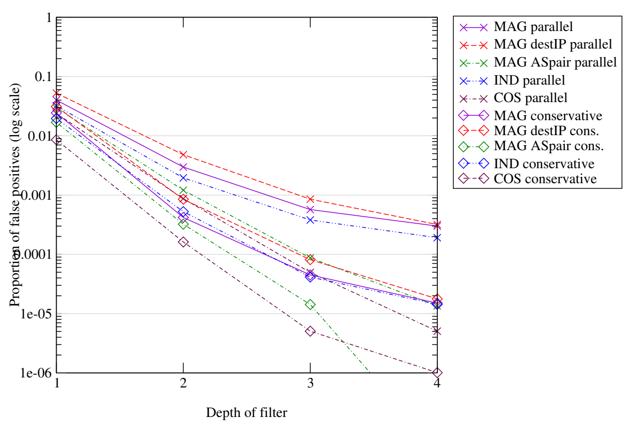

Fig. 17. Conservative update for a stage strength of k=3

by conservative update grows to more than an order of magnitude as the number of stages increases. For the configuration with 4 stages of strength 3 we obtained no false positives when running on the MAG trace with flows defined by AS pairs and that is why the plotted line "falls off" so abruptly. Since by extrapolating the curve we would expect to find approximately 1 false positive, we consider that this data point does not invalidate our conclusions.

C.3 The effect of preserving entries and shielding

Our next set of experiments evaluates the combined effect of preserving entries and shielding. We run experiments with filter depths from 1 to 4 and stage strengths of 0.5 and 2. We measure the largest number of entries of flow memory used and the

Trace + flow definition	Error when preserving entries
MAG 5-tuple	19.12% - 26.24%
MAG destination IP	23.50% - 29.17%
MAG AS pairs	16.44% - 17.21%
IND 5-tuple	23.46% - 26.00%
COS 5-tuple	30.97% - 31.18%

Table IX. Average error when preserving entries as percentage of the average error in the base case

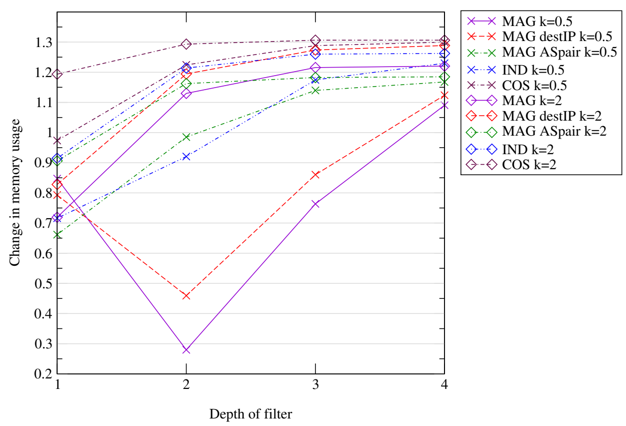

Fig. 18. Change in memory usage due to preserving entries and shielding

average error of the estimates. When computing the maximum memory requirement we ignored the first two measurement intervals in each experiment because the effect of shielding is fully visible only from the third measurement interval on.

The improvement in the average error does not depend much on the filter configuration. Table IX shows the results for each trace and flow definition. Usually for the weak filters (few, weak stages) the reduction in the average error is slightly larger than for the strong ones.

There are two conflicting effects of preserving entries on the memory requirements. On one hand by preserving entries we increase the number of entries used. On the other hand shielding increases the strength of the filter (see Section 4.2.3 for details) which leads to a decrease in the number of false positives. Figure 18 shows how memory usage is influenced by preserving entries. The first effect predominates for strong filters leading to an increase in memory usage by up to 30%. The second one predominates for weak filters leading to a decrease by as much as 70%.