New Directions in Traffic Measurement and Accounting

Cristian Estan and George Varghese

February 8, 2002

Abstract

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 log 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 on measuring only large flows — 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.

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 [9, 5]. Such measurement information is essential for short-term monitoring (e.g., detecting hot spots and denial-of-service attacks [15]), longer term traffic engineering (e.g., rerouting traffic and upgrading selected links[9]), and accounting (e.g., to support usage based pricing[6]).

The standard approach advocated by the Real-Time Flow Measurement (RTFM) [4] 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 [17] 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 [21] showed over 250,000 concurrent flows. More recent measurements in [8] 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 [8] 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 the counters in DRAM is already impossible with today's line speeds and the gap between DRAM speeds (improving 7-9% per year) and link speeds (improving 100% per year) is only going to increase. Cisco NetFlow [17], which keeps its flow counters in DRAM solves this problem by sampling: only the sampled packets result in updates. But this sampling has problems of its own (as we show later) since it affects the accuracy of the measurement data.

Despite the large number of flows, a common observation found in many measurement studies (e.g., [9, 8]) is that a small percentage of flows accounts for a large percentage of the traffic. [8] shows that the top 9% of the flows between AS pairs accounts for 90% of the traffic in bytes between all AS pairs.

For many applications, knowledge of these large flows is most important. [8] suggests that scalable differentiated services could be achieved by providing selective treatment only to a small number of large flows or aggregates. [9] underlines the importance of knowledge of "heavy hitters" for decisions about network upgrades and peering. [6] 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 not feasible to accurately measure all flows on high speed links, but many applications can benefit from accurately measuring the few large flows that dominate the traffic mix. This can be achieved by traffic measurement devices that use small fast memories. However, how does the device know which flows to track? If one keeps state for all flows to identify the heavy hitters, our purpose is defeated.

Thus a reasonable goal is to produce an *algorithm that identifies the heavy hitters using memory that is only a small constant larger than what we need to track the heavy hitters.* This is the central question addressed by this paper. We present two algorithms that identify the large flows using a small amount of state. Further, we have low worst case bounds on the amount of per packet processing, making our algorithms suitable for use in high speed routers.

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)[1]. 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.

Large flows are defined as those that send more than a given threshold (say 1% of the link capacity) during a given measurement interval (1 second, 1 minute or even 1 hour). Appendix C gives an alternative definition of large flows based on leaky bucket descriptors, and investigates how our algorithms can be adapted to this definition.

An ideal algorithm reports, at the end of the measurement interval, the flow IDs of all the flows that exceeded the threshold and their exact size. There are three ways in which the result can be wrong: it might omit some of the large flows, it might erroneously add some small flows to the report or it might 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*.

[1] 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)

Note that the minimum amount of memory required by an ideal algorithm is the inverse of the threshold; for example, there can be at most 100 flows that use more than 1% of the link. We will measure the performance of an algorithm by its memory (compared to that of the ideal algorithm), and the probability of false negatives and false positives.

1.2 Motivation

Our algorithms for identifying large flows can potentially be used to solve many problems. 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 of access or a fixed price per month, regardless of how much the flow transmits). While usage-based pricing [14, 20] has been shown to improve overall utility by providing incentives for users to reduce traffic, 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, 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 can offer a compromise between the two extremes that is scalable and yet offers almost the same utility as usage based pricing. [1] offers experimental evidence based on the INDEX experiment that such threshold pricing could be attractive to both users and ISPs. [2].
- **Real-time Traffic Monitoring:** Many ISPs monitor their backbones to look for hot-spots. Once a hot-spot is detected one would want to identify the large aggregates that could be rerouted (using MPLS tunnels or new routes through reconfigurable optical switches) to alleviate congestion. Also ISPs might want to monitor traffic to detect (distributed) denial of service attacks. Sudden increases in the traffic sent to certain destinations (the victims) can indicate an ongoing attack. [15] proposes a mechanism that reacts to them as soon as they are detected. In both these settings, it may be sufficient to focus on flows above a certain traffic threshold.
- **Scalable Queue Management:** As we move further down the time scale, there are other applications that would benefit from identifying large flows. Scheduling mechanisms aiming to ensure (weighted) max-min fairness (or an approximation thereof), need to be able to detect the flows sending above their fair rate and penalize them. Keeping per flow state only for these flows does not affect the fairness of the scheduling and can account for substantial savings. This problem is actually more complicated because the definition of a non-conformant flow can depend on round-trip delays as well. Several papers address this issue including [10]. We do not address this application further in the paper, except to note that our techniques may be useful as a component in solutions to this problem.

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] Besides [1], a brief reference to a similar idea can be found in [20]. However, neither paper proposes a corresponding mechanism to implement the idea at backbone speeds. [6] offers a mechanism to implement threshold accounting that is suitable if the timescale for billing is long.

2 Related work

The primary tool used for flow level measurement by IP backbone operators is Cisco NetFlow [17] (see Appendix E for a more detailed discussion). NetFlow keeps per flow state 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. [9] 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. We show later that 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. 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, [8] 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 [16] produced 0.5GB per hour of compressed NetFlow data and aggregation reduced this data only by a factor of 4. Techniques described in [6] 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 paper address the problem of mapping the traffic of large IP networks. [9] 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. [5] describes a mechanism for identifying packet trajectories in the backbone, not identifying the networks generating the traffic.

Bloom filters [2] and stochastic fair blue [10] use similar but different techniques to our parallel multistage filters to compute different metrics (set intersections and drop probabilities). Gibbons and Matias [11] consider synopsis data structures that use small amounts of memory to approximately summarize large databases. They define counting samples that are similar to our sample and hold algorithm. However, we compute a different metric, need to take into account packet lengths and have to size memory in a different way. In [7], Fang et al look at efficient ways of exactly counting the number of appearances of popular items in a database. Their multi-stage algorithm is similar to the multistage filters 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.

3 Our solution

Because our algorithms use an amount of memory that is a constant factor larger than the (relatively small) number of heavy-hitters, 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)[3] allocate a new entry if there is no entry associated with the current packet.

The biggest problem is to identify the large flows. Two simple approaches to identifying large flows suggest themselves immediately. First, when a packet arrives with a flow ID not in the flow memory, we

[3]Furthermore, the improvement presented in Appendix E that can be applied to NetFlow and our algorithms increases by an order of magnitude the amount of time we can spend on a packet

could make place for the new flow by removing the flow with the smallest measured traffic (i.e., smallest counter). It is easy, however, 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.

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. We show, however, in Table 1 that 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 for multistage filters a slightly 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 for the same amount of memory than random sampling. In Section 3.3 we present improvements to the two algorithms that further improve their accuracy on actual 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 belonging to the flow as shown in Figure 1.

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[4]. Choosing a high enough value for p guarantees that flows above the threshold are very likely to be detected. Increasing p too much 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 the analysis more concretely. Suppose we wish to measure the traffic sent by all the flows that take over 1% of the link capacity in a measurement interval. There are at most 100 such flows that take over 1%. 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 end of the measurement interval (the probability of a 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

[4] We actually sample packets, but the sampling probability depends on packet sizes. The sampling probability for a packet of size s is $p_s = 1 - (1-p)^s$. This can be looked up in a precomputed table or approximated by $p_s = p * s$.

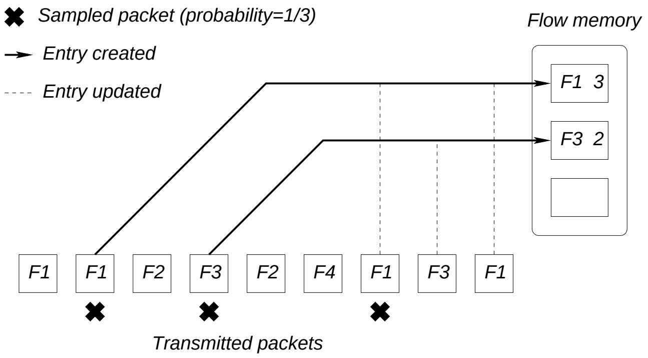

Figure 1: The leftmost packet with flow label $F1$ arrives first at the router. After an entry is created for a flow (solid line) the counter is updated for all its packets (dotted lines)

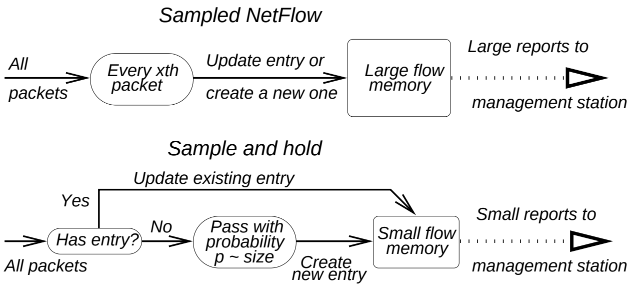

Figure 2: Sampled NetFlow counts only sampled packets, sample and hold counts all after entry created

oversampling factor. Better still, the probability that flow F is in the flow memory after sending 5% of its traffic is, using a similar analysis, $1 - e^{-5}$ 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 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 that the average number of flows sampled is no more than 10,000 but some intervals can sample more packets. However, the distribution of the number of samples is binomial with a small standard deviation equal to the 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.

When compared to Cisco's sampled NetFlow our idea has three significant differences depicted in Figure 2. 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 section 5 this will make our results much more accurate. Second, our sampling technique avoids packet size biases unlike NetFlow which samples

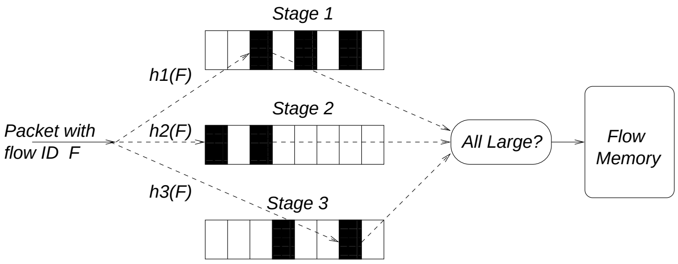

Figure 3: In a parallel multistage filter, a packet with a flow ID F is hashed using hash function $h1$ into a Stage 1 hash table, $h2$ into a Stage 2 hash table, etc. Each of the hash buckets contain a counter that is incremented by the packet size. If *all* the hash bucket counters are above the threshold (shown bolded), then flow F is passed to the flow memory for more careful observation.

every x packets. Third, our technique avoids the extra resource overhead (router processing, router memory, network bandwidth) of sending the large amount of sampled information to a management station (assuming only information about heavy-hitters will be used at the 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 if it had 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).

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 3,1, 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 - 100Kbytes = 900 Kbytes.

[5]To simplify computation, in our examples we assume that 1Mbyte=1,000,000 bytes and 1Kbyte=1,000 bytes.

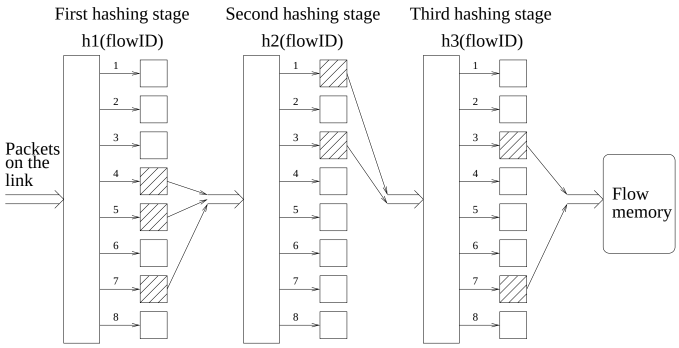

Figure 4: Serial multistage filter: packets that hash to large buckets are passed to the next stage

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 100Kbytes passing the filter is at most $100,000 * 15.2 * 10^{-4} < 16$. There can be at most 999 flows above 100Kbytes, 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 at packet arrival time performed by the filter is exactly one read and one write per stage. If the number of stages is small enough this is affordable even at high speeds since the memory accesses can be performed in parallel, especially in a chip implementation.[6] While multistage filters are more complex than sample-and-hold, they have a number of advantages. They reduce the probability of false negatives to 0 and by decreasing the probability of false positives, they reduce the size of the required flow memory.

3.2.1 The serial multistage filter

In this section we briefly present another variant of the multistage filter called a serial multistage filter (Figure 4). Instead of using multiple stages in parallel, we can put them after each other, each stage seeing only the packets that passed the previous stage (and all stages preceding it).

Let d be the number of stages (the depth of the serial filter). We set a 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 negatives. As with parallel filters, small flows can pass the filter only if they are lucky enough to hash to

[6] We describe details of a preliminary OC-192 chip implementation of multistage filters in Section 8.

counters with significant traffic generated by other flows.

The analytical evaluation of serial filters is somewhat more complicated than for parallel filters. Since, as presented 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 improve 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 Preserving entries across measurement intervals

Measurements show that large flows also tend to last long. Applying our algorithms directly would mean erasing the flow memory after each interval. This means that in each interval, the bytes of large flows sent before they are allocated an entry are not counted. By preserving the entries of large flows across measurement intervals and only reinitializing the counters, only the first measurement interval has this inaccuracy, so *all long lived large flows are measured exactly.*

The problem is that the algorithm cannot distinguish between a large flow that was identified late and a small flow that was identified by error since both have small counter values. A conservative solution is to preserve the entries of not only the flows for which we count at least T bytes transferred in the current interval, but all the flows whose entries were added in the current interval (since their traffic might be above T if we also add their traffic that went by before the flow was identified). While more complex rules for which entries to keep can be devised, we found little advantage in most of them and therefore do not discuss them here. The next section presents a rare exception.

3.3.2 Early removal of entries

While the simple rule for preserving entries described above works well for both of our algorithms, there is a refinement that can help further in the case of sample and hold which has a more false positives than multistage filters. If we keep for one more interval all of the flows that got 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.

3.3.3 Shielding the filter from flows with entries

Shielding strengthens multistage filters. Figure 5 illustrates how it works. The traffic belonging to flows that have an entry no longer passes through the filter. It may not be immediately apparent how this reduces the number of false positives. Consider large, long lived flow that would go through the filter each measurement interval. Each measurement interval, the counters it hashes to exceed the threshold. If we shield the filter from this large flow, many of these counters will not reach the threshold after the first interval. This reduces the probability that a random small flow passes the filters by hashing to counters that are large because of other flows.

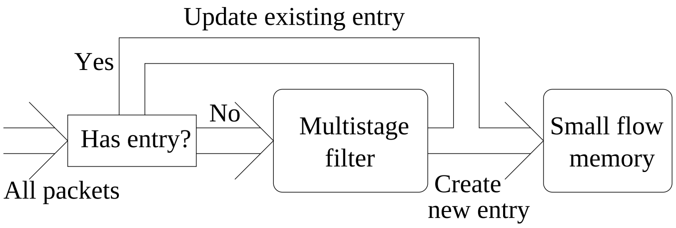

Figure 5: Shielding: we do not pass through the filter the traffic of the flows with an entry

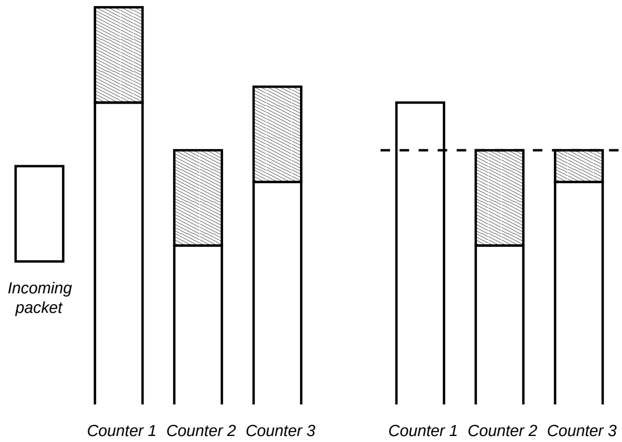

Figure 6: Conservative update: without conservative update (left) all counters are increased by the size of the incoming packet, with conservative update (right) no counter is increased to more than the size of the smallest counter plus the size of the packet

3.3.4 Conservative update of counters

We now describe an important but natural optimization for multistage filters. *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 6) 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* (counters are never decremented). 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.

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 $T-x$ and the size of the packet is $s > x > 0$. Normally one would increment the counter at stage i to T 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 $s-x < T$). Since the counter at stage i was below T, 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 was allowed to change during a measurement interval.

4 Analytical evaluation of our algorithms

In this section we analytically evaluate our algorithms. 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 (based on its analysis from appendix E).

Example: We will use the following running example to give numeric instances for the analysis. Assume a 100 Mbyte/s link with 100,000 flows. We want to identify and measure all flows whose traffic is more than 1% (1 Mbyte) of the link capacity during 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 running example, this would mean setting p to 1 in 50,000 bytes for an oversampling of 20 and 1 in 200,000 for an oversampling of 5. With an average packet size of 500 bytes this is roughly 1 in 100 packets and 1 in 400 packets respectively.

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. 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 of the estimate of the size of a flow at the threshold is 7% and with an oversampling of $O = 5$ 28%. Applying the correction would bring down the errors to 5% and 20% respectively.

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 and an oversampling of 5 500 entries.

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 packets is $\sqrt{Cp(1-p)}$.

Example: For our example for an oversampling of 20 and an overflow probability of 0.1% we need at most 2,147 entries and with an oversampling of 5, 574 entries. If the acceptable overflow probability is 1%, the sizes are 2,116 and 558 respectively.

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.

[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 our 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).

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[9]. 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 the number of sampled packets during the 2 intervals which is $\sqrt{2Cp(1-p)}$.

Example: For our example with an oversampling of 20 and acceptable probability of overflow equal to 0.1%, the flow memory has to have at most 4,207 entries and with an oversampling of 5, 1,104 entries. If the acceptable overflow probability is 1%, the sizes are 4,164 and 1,082 respectively.

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 and from 0.67% to 1.8% with an oversampling of 5.

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 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 a flow memory with 2,647 entries and with an oversampling of 5, 1,074 entries. If the acceptable overflow probability is 1%, the sizes are 2,616 and 1,058 respectively.

4.2 Multistage filters

In this section, we analyze parallel multistage filters. We only present the main results. The proofs and supporting lemmas are in Appendix A. 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 expresses the strength of the filtering achieved by a stage of the filter: 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 can also be seen as the memory over-provisioning ratio: by what factor do we inflate each stage memory beyond the required minimum of C/T?

[9]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 such flow, because if there were, many of their packets would be sampled, decreasing the number of entries required.

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. 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 that go 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.

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.

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.

This is a bound on the *expected* number of flows passing. In Appendix A we derive a high probability bound on the number of flows passing the filter..

Example: The probability that more than 185 flows pass the filter is at most 0.1% and the probability that more than 211 pass is no more than $1 - 10^{-6}$. Thus by increasing the flow memory from the expected size of 122 to 185 we can make overflow of the flow memory extremely improbable.

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.

5 Comparison of traffic 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 state 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 comaparisons are based on the algorithm analysis in Section 4 and an analysis of NetFlow from Appendix E.

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. We focus on the accuracy of the measurement of a 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 1.

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

Measure	Sample and hold	Multistage filters	Sampling
Relative error	$\frac{\sqrt{2}}{Mz}$	$\frac{1+10\ r\ log_{10}(n)}{Mz}$	$\frac{1}{\sqrt{Mz}}$
Memory accesses	1	$1 + log_{10}(n)$	$\frac{1}{x}$

Table 1: Comparison of the core algorithms: sample and hold provides most accurate results while pure sampling has very few memory accesses

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	$2O/z$	$2/z + 1/zlog_{10}(n)$	min(n,486000 t)
Memory accesses	1	$1\ + log_{10}(n)$	$1/x$

Table 2: Comparison of traffic measurement devices

so that a single small flow is expected to pass the filter[10] and the strength of the stages is 10. At this point we estimate the memory usage to be $M = b/k + 1 + rbd = C/T + 1 + r10C/Tlog_{10}(n)$ where r 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 the error of our estimate of flows of size zC and the result from Table 1 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 1 gives the number of memory accesses per packet that each algorithm performs. Since sample and hold performs a packet lookup for every packet[11], its per packet processing is 1. Multistage filters add to the one flow memory lookup an extra one access per stage; the number of stages in turn increases as the logarithm of the number of flows. Finally, for ordinary sampling if one in x packets get sampled, then the average per packet processing is $1/x$.

Table 1 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 scale 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 (< 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.

[10] 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.

[11] 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.

5.2 Comparison of traffic measurement devices

Table 1 seems to imply that if we increase the DRAM memory size M to infinity, we can make the relative error of a Sampled NetFlow estimate arbitrarily small. Intuitively, this assumes that by increasing memory one can increase the sampling rate so that x decreases to become arbitrarily close to 1. Clearly, if $x = 1$, the results for Sampled NetFlow would, of course, have no error since every packet is logged. But we have just seen that x must at least be as large as the ratio of DRAM speed to SRAM speed; thus Sampled NetFlow will always have a minimum error corresponding to this value of x.

Another way to see the same effect is to realize that for a fixed value of x, there is a limit M' to the amount of DRAM memory that can be accessed during a measurement interval. In the worst case, the number of packets sampled by ordinary sampling is M' out of at most C/y_{min} packets, where C is the link capacity and y_{min} is the minimum size for a packet. Thus $x = C/(y_{min}M')$ and so $M' = C/(xy_{min})$. *Thus increasing M beyond M' does not help!*

With this as the basic insight, we now compare the performance of our algorithms and NetFlow in Table 2 without limiting the amount of memory made available to NetFlow. Table 2 takes into account the underlying technologies (i.e., the use of DRAM versus SRAM) and one optimization (i.e., preserving entries) for both 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[12]. 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). 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[13]. 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 (based on its historically slow pace of progress compared to chip speeds).

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 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 (Table 1) that stem from the core algorithms, we see new differences in Table 2. The first big difference (Row 1 of Table 2) is that unlike NetFlow, *our algorithms provide exact measures for long-lived large flows* by preserving entries. More precisely, by preserving entries our algorithms will exactly

[12] 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.

[13] We assume that a flow memory entry is equivalent to 10 of the counters used by the filter because the flow ID is typically much larger than the counter.

[14] If the capacity of the link is x times OC-3, then one in x packets gets sampled. We assume based on [17] that NetFlow can handle packets no smaller than 40 bytes at OC-3 speeds.

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, this is a big advantage.[15]

The second big difference (Row 2 of Table 2) is that we can make our algorithms arbitrarily accurate at the cost of increases in the amount of memory used[16] while sampled NetFlow can do so only by increasing the measurement interval t.

The third row of Table 2 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 2 is identical to the second row of Table 1.

Table 2 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 2. These are **iii)** provable lower bounds on traffic, **iv)** reduced resource consumption for collection, and **v)** faster detection of new large flows. We briefly examine these dvantages.

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 average overcharged amount arbitrarily low (using large measurement intervals), 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 much smaller logs, our algorithm avoids these resource (e.g., memory, transmission bandwidth, and router CPU cycles) bottlenecks.

v) Faster detection of long-lived flows: In a security or DoS application, it may be useful to quickly detect a large increase in traffic to a server. Our algorithms can use small measurement intervals and detect large flows soon after they start. By contrast, Sampled NetFlow, especially when mediated through a management station, can be much slower.

6 Dimensioning traffic measurement devices

Before we describe measurements, we describe how to dimension our two 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 that hold for any ditribution of flow sizes. When we can make some assumptions about the distribution of flow sizes, we can arrive to some tighter bounds as in Appendix B does for the case of a Zipf distribution. Section 7 shows that the performance of our algorithms on actual traces exceeds as much as tens of thousands of times our conservative analysis. Dimensioning according to the safe, conservative bounds can be a waste resources for applications such as measurement for accounting purposes, where the

[15] 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 log 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.

[16] 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 to make this not be an issue.

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

Figure 7: The threshold adapts dynamically to achieve the target memory usage

ability to handle adversarial behavior is less important than the overall accuracy of the results. In this section we look at more aggressive methods of configuring the traffic measurement devices that maximize the accuracy of the results by making good use of the available memory.

The measurements from section 7 show that the actual performance depends strongly on the traffic mix. Since we usually don't have a priori knowledge of flow distributions, we prefer to dynamically adapt algorithm parameters to actual traffic. The main idea we use is to *keep decreasing the threshold below the conservative estimate until the flow memory is nearly full* (totally filling memory can result in new large flows not being tracked). We only discuss here the algorithm used for adapting the threshold.Appendix D gives the heuristics we use to set the configuration parameters for the multistage filters that are hard to adapt dynamically to the traffic (i.e. the number of counters and stages).

Figure 7 presents our threshold adaptation algorithm. There are two important constants that adapt the threshold to the traffic: the "target usage" (variable $target$ in Figure 7) that tells it how full the memory can be without risking to fill it up completely and the "adjustment ratio" (variables $adjustup$ and $adjustdown$ in Figure 7) that the algorithm uses to decide how much to adjust the threshold to achieve a desired increase or decrease in flow memory usage. We rely on the measurements from Appendix I to determine the actual values for these constants.

The usage of the flow memory oscillates even when the configuration is fixed. This happens due to changes in the traffic mix or simply due to the randomness of our algorithms. The measurements from Appendix I determine how volatile the number of entries used is and based on them, set the target usage to 90% for both algorithms.

One can argue that intuitively the number of entries should be proportional to the inverse of the threshold since the number of flows that can exceed a given threshold is inversely proportional to the value of the threshold. This corresponds to having an adjustment ratio of 1. In practice it might happen that increasing the threshold does not reduce the number of used entries by very much because fewer flows than expected are between the two values of the threshold. On the other hand decreasing the threshold can cause a collapse of the multistage filter increasing very much the number of flows that pass. To give robustness to the traffic measurement device we use two different adjustment ratios: when increasing the threshold we use a large one (we conservatively assume that we need to increase the threshold by $adjustup$% to decreases memory usage by only 1%) and when decreasing we use a small one (we conservatively assume that decreasing the threshold by only $adjustdown$% we increase the memory usage by 1%). We use measurements to bound from above and below the effect of the changes of threshold on the number of memory entries used and derive

Trace	Number of flows (min/avg/max)			Mbytes/interval (min/avg/max – link)
	5-tuple	destination IP	AS pair	
MAG+	93,437/98,424/105,814	40,796/42,915/45,299	7,177/7,401/7,775	201.0/256.0/284.2 – 1483
MAG	99,264/100,105/101,038	43,172/43,575/43,987	7,353/7,408/7,477	255.8/264.7/273.5 – 1483
IND	13,746/14,349/14,936	8,723/8,933/9,081	-	91.37/96.04/99.70 – 370.8
COS	5,157/5,497/5,784	1,124/1,146/1,169	-	14.28/16.63/18.70 – 92.70

Table 3: The traces used for our measurements

the adjustment ratios. Based on the measurements from Appendix I, we use a value of 3 for *adjustup*, 1 for *adjustdown* in the case of sample and hold and 0.5 for multistage filters.

To give further 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. If the threshold decreased within the last 3 measurement intervals we conservatively consider only the memory usage values recorded with the low threshold. Since changes of the threshold take 2 measurement intervals to fully show their effects on the memory usage we consider that using a window of 3 measurement intervals to average over is a good tradeoff between responsiveness to changes in the traffic mix and fast convergence to a good value for the threshold.

7 Measurements

Performance cannot be evaluated solely through the use of Zen Meditation. (paraphrased from Jeff Mogul)

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 the "clear" trace MAG+ and we refer to them 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

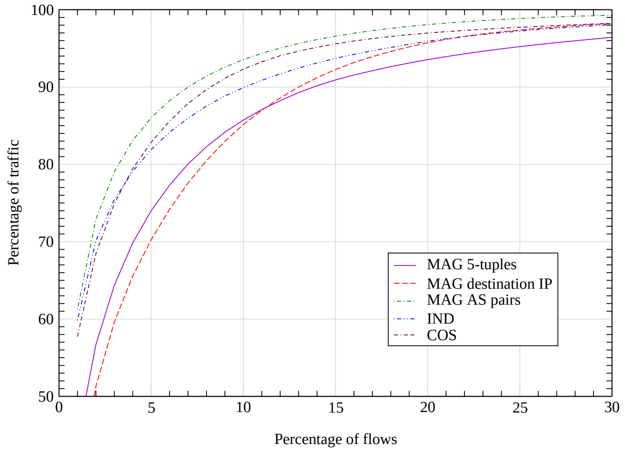

Figure 8: Cumulative distribution of flow sizes for various traces and various flow definitions

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 3 gives a summary description of 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. Appendix F gives the measurements we base this decision on. Here we only note that in all cases 99% or more of the 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 8 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 from the figure, 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

We present detailed measurements on the performance on sample and hold in and its optimizations in Appendix G. The detailed results for multistage filters are in Appendix H. 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.

Algorithm	Maximum memory usage / Average error				
	MAG 5-tuple	MAG destination IP	MAG AS pair	IND 5-tuple	COS 5-tuple
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 / 24.33%	1,964 / 24.07%	714 / 24.40%	1,313 / 23.83%	710 / 22.17%
+ preserve entries	3,832 / 4.67%	3,213 / 3.28%	1,038 / 1.32%	1,894 / 3.04%	1,017 / 6.61%
+ early removal	2,659 / 3.89%	2,294 / 3.16%	803 / 1.18%	1,525 / 2.92%	859 / 5.46%

Table 4: 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 4 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.[17]

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 impressive 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 (our measurements indicate that 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 preserving entries caused it to increase most.

We performed measurements on many more configurations, but for brevity we report them only in Appendix G. The results are in general similar to the ones from Table 4, 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

[17] We defined the relative error by dividing the average error by the size of 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.

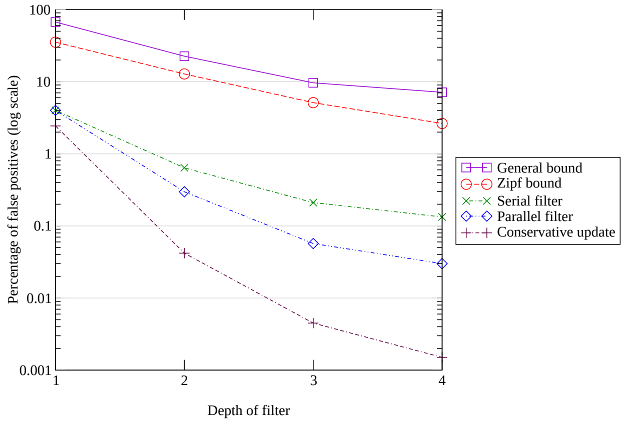

Figure 9: Filter performance for a stage strength of k=3

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 9 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.

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. Preserving entries reduces the average error in the estimates by 70% to 85%. Its effect depends 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 flow memory entries by as much as 70%.

7.2 Evaluation of complete traffic measurement devices

In this section we 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[18]) 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 hashing 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 Appendix D, we use 3114 counters[19] 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 a hundredth of a thousandth – 15552 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 moduli 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. For the unidentified flows, we consider that the error is equal to their total traffic. Tables 5 to 7 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.

We believe these results (and similar results not presented here for lack of space) do confirm that our algorithms are better than sampled NetFlow at measuring the largest of the flows. The results for multistage filters are always slightly better than those for sample and hold despite the fact that we use fewer memory locations because we have to sacrifice part of the memory for the counters of the stages. We do not consider this to be a definitive proof of the superiority of multistage filters, since tighter algorithms for adapting the threshold can possibly result in further improvements of the performance of 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).

[18] 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).

[19] 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%
$0.1 \ldots 0.01\%$	1.797% / 7.086%	0% / 1.090%	0.02132% / 22.02%
$0.01 \ldots 0.001\%$	77.01% / 61.20%	54.70% / 43.87%	17.72% / 50.27%

Table 5: 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%
$0.1 \ldots 0.01\%$	0.4289% / 3.153%	0% / 0.9488%	0.01381% / 20.77%
$0.01 \ldots 0.001\%$	65.72% / 51.19%	49.91% / 39.91%	11.54% / 46.59%

Table 6: Comparison of traffic measurement devices with flow IDs defined by destination IP

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 7: Comparison of traffic measurement devices with flow IDs defined by the source and destination AS

8 Implementation Issues

In this section we briefly describe implementation issues for the two algorithms. Sample and Hold is fairly straightforward to implement even in a network processor because it adds only one memory reference to packet processing, assuming there is sufficient SRAM for flow memory and assuming an associative memory. For small flow memory sizes, adding a CAM is quite feasible. Alternatively, one can implement an associative memory using a hash table and storing all flow IDs that collide in a much smaller CAM. Sample and Hold does require a source of random numbers but most routers require this anyway to implement algorithms such as RED.

Multistage filters are harder to implement using a network processor because they need multiple memory references (to stage memories) in addition to the associative lookup of flow memory. However, multistage filters are fairly easy to implement in an ASIC as the following feasibility study shows. [12] describes a chip designed to implement a parallel multistage filter with 4 stages of 4K counters[20] each and a flow memory[21] of 3584 entries. The chips runs at OC-192 line speeds: it accepts a header every 32 nanoseconds. It has a cycle time of 8ns. Each entry in the flow memory is 27 bytes wide and contains the flow ID, number of bytes and packets and the timestamp of the first and last packet. The chip has an interface to a management processor that can read and write the flow memory. The core logic of the chip consists of approximately 450,000 transistors that fit on 2mm x 2mm on a .18 micron process. The hash stage counters would occupy a further 8.4 mm^2 and the flow memory takes 21 mm^2. Including the memories and the 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. Both the size and the power 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 1 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 1. This is because Sampled NetFlow does not process every packet and hence can afford to use large DRAM. Despite this, results in Table 2 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. NetFlow records allow managers to *a posteriori* mine 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)?

[20] The counters are on 32 bits.

[21] Entries are located in the flow memory with the help of 3 hash functions in the manner described in [3].

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 practice.

We end by noting that the measurement problems faced by network managers are extremely similar to the measurement problems faced by other areas in computer science such as data mining, architecture, and even compilers. For example, Jim Smith and his co-workers [19] recently proposed using a Sampled NetFlow-like strategy to obtain dynamic instruction profiles in a processor (which are used for later optimization). We have preliminary results that show that the use of multistage filters with conservative update can improve the results of [19] for determining instruction profiles. 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.

References

[1] Jorn Altman and Karyen Chu. A proposal for a flexible service plan that is attractive to users and internet service providers. In *IEEE Proceedings of the INFOCOM*, April 2001.

[2] B. Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, volume 13, pages 422–426, July 1970.

[3] Andrei Z. Broder and Anna R. Karlin. Multilevel adaptive hashing. In *Proceedings of ACM-SIAM symposium on Discrete algorithms*, pages 43–53, January 1990.

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

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

[6] Nick Duffield, Carsten Lund, and Mikkel Thorup. Charging from sampled network usage. In *SIGCOMM Internet Measurement Workshop*, November 2001.

[7] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. In *International Conference on Very Large Data Bases*, pages 307–317, August 1998.

[8] Wenjia Fang and Larry Peterson. Inter-as traffic patterns and their implications. In *Proceedings of IEEE GLOBECOM*, December 1999.

[9] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving traffic demands for operational ip networks: Methodology and experience. In *Proceedings of the ACM SIGCOMM*, pages 257–270, August 2000.

[10] Wu-chang Feng, Dilip D. Kandlur, Debanjan Saha, and Kang G. Shin. Stochastic fair blue: A queue management algorithm for enforcing fairness. In *IEEE Proceedings of the INFOCOM*, April 2001.

[11] Phillip B. Gibbons and Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the ACM SIGMOD*, pages 331–342, June 1998.

[12] John Huber. Design of an oc-192 flow monitoring chip. Class Project, March 2001.

[13] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of the ACM SIGCOMM*, pages 203–214, September 1998.

[14] J. Mackie-Masson and H. Varian. *Public Access to the Internet*, chapter Pricing the Internet. MIT Press, 1995.

[15] Ratul Mahajan, Steve M. Bellovin, Sally Floyd, John Ioannidis, Vern Paxson, and Scott Shenker. Controlling high bandwidth aggregates in the network. http://www.aciri.org/pushback/, July 2001.

[16] David Moore. Personal conversation. also see caida analysis of code-red, 2001. `http://www.caida.org/analysis/ security/ code-red/`.

[17] Cisco netflow. `http://www.cisco.com /warp /public /732 /Tech /netflow`.

[18] Sampled netflow. `http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120limit/120s/120s11/12s_sanf.htm`.

[19] Subramanya Sastry, Ratislav Bodik, and James E. Smith. Rapid profiling via stratified sampling. In *28th. International Symposium on Computer Architecture*, pages 278–289, June 2001.

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

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

A Details of the analytical evaluation of multistage filters

This section presents the detailed analytical evaluation of parallel multistage filters. We use the same notation as in section 4.2. We first derive the bound for the expected number of flows passing the filter. After that we give two high probability bounds on the number of flows passing the filter: a loose bound that has a closed form and a tighter one we specify as and algorithm.

Lemma 4 *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 $T-s$. 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 $T-s$ 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 5 (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 4 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. $\blacksquare$

Before using this lemma to derive a bound on the number of flows passing a multistage filter, we can use it for binding from below the expected error in the estimate of the size of a large flow.

Corollary 5.1 *For a flow with size $s > T$ and no packets larger than y_max, the probability that the number of undetected bytes $s-c$ is at least x is bound by $P(s-c \geq x) \leq \left(\frac{1}{k}\frac{T}{T-x-y_max+1}\right)^d$*

Proof There is a sequence of packets at the beginning of the flow of length $x \leq s_s \leq x + y_max - 1$. If this sequence does not pass the filter than $s-c \geq s_s \geq x$. By lemma 1 we can bound this probability and this gives us this corollary. $\blacksquare$

Theorem 6 (2) *The expected number of bytes of a large flow that go undetected by a multistage filter is bound from below by*

$$E[s-c] \geq T\left(1 - \frac{d}{k(d-1)}\right) - y_max \quad (3)$$

Proof

$$
\begin{aligned}
E[s-c] &= \sum_{x=0}^{T-1} P(s-c=x)x = \sum_{x=1}^{T-1} P(s-c=x)x = \sum_{x=1}^{T-1} P(s-c \geq x) \geq \sum_{x=1}^{T\frac{k-1}{k}-y_max} P(s-c \geq x) \\
&= \sum_{x=1}^{T\frac{k-1}{k}-y_max} 1 - P(s-c<x) = T\frac{k-1}{k} - y_max - \sum_{x=1}^{T\frac{k-1}{k}-y_max} P(s-c<x) \\
&\geq T\left(1-\frac{1}{k}\right) - y_max - \sum_{x=1}^{T\frac{k-1}{k}-y_max} P(s-c \leq x)
\end{aligned}
$$

Through corollary 5.1 we can give an upper bound for the sum.

$$
\begin{aligned}
\sum_{x=1}^{T\frac{k-1}{k}-y_max} P(s-c \leq x) &\leq \sum_{x=1}^{T\frac{k-1}{k}-y_max} \left(\frac{1}{k}\frac{T}{T-x-y_max+1}\right)^d \leq \int_{x=1}^{T\frac{k-1}{k}-y_max+1} \left(\frac{1}{k}\frac{T}{T-x-y_max+1}\right)^d dx \\
&= \left(\frac{T}{k}\right)^d \int_{x=1}^{T\frac{k-1}{k}-y_max+1} \left(\frac{1}{T-y_max+1-x}\right)^d dx \\
&= \left(\frac{T}{k}\right)^d \frac{1}{d-1}\left(\frac{1}{T-y_max+1-x}\right)^{d-1} \Big|_{x=1}^{T\frac{k-1}{k}-y_max+1} \\
&\leq \frac{(T/k)^d}{d-1}\left(\frac{1}{T-y_max+1-T\frac{k-1}{k}+y_max-1}\right)^{d-1} = \frac{(T/k)^d}{d-1}\left(\frac{1}{T/k}\right)^{d-1} = \frac{T}{k(d-1)}
\end{aligned}
$$

By substituting this result we obtain $E[s-c] \geq T\left(1-\frac{1}{k}\right) - y_max - \frac{T}{k(d-1)} = T\left(1-\frac{d}{k(d-1)}\right) - y_max$. ■

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

Theorem 7 (3) *The expected number of flows passing a parallel multistage filter is bound by*

$$
E[n_{pass}] \leq max\left(\frac{b}{k-1}, n\left(\frac{n}{kn-b}\right)^d\right) + n\left(\frac{n}{kn-b}\right)^d \tag{4}
$$

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 $E[n_{i_{pass}}] = n_i p_{s_i} \leq n_i\ max(1, (\frac{1}{k}\frac{T}{T-s_i})^d)$. 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 $s_i > T\frac{k-1}{k}$. For these $E[n_{i_{pass}}] \leq n_i = \frac{h_i C}{s_i} < \frac{h_i C}{T\frac{k-1}{k}}$, therefore substituting them 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 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 $E[n_{small_{pass}}] \leq np_{\frac{C}{n}}$.

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

Now we will 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 $CT^d f(T\frac{k-1}{k}) = \frac{C}{T\frac{k-1}{k}} = \frac{b}{k-1}$ or $CT^d f(\frac{C}{n}) = n(\frac{1}{k}\frac{T}{T-\frac{C}{n}})^d = n(\frac{n}{kn-b})^d$. Substituting these values we obtain the bound. ■

For proving our high probability bounds, we use the following result from probability theory.

Lemma 8 *Assume we have a sequence of n independent events succeeding with probability p. The probability that the number of events suceeding i exceeds the expected value by more than λ is bound by*

$$
Pr(i > np + \lambda) \le e^{-\frac{\lambda^2}{2np + \frac{2}{3}\lambda}}
$$

Corollary 8.1 *If we want to limit the probability of underestimation to p_{safe} for the experiment above we can bound i by*

$$
i \le \lfloor np - \frac{ln(p_{safe})}{3} + \sqrt{\frac{ln(p_{safe})^2}{9} - 2np\ ln(p_{safe})} \rfloor
$$

Proof By lemma 8 we have

$$
e^{-\frac{\lambda^2}{2np + \frac{2}{3}\lambda}} \le p_{safe}
$$

We can determine λ by solving the resulting quadratic equation.

$$
\lambda^2 + \frac{2ln(p_{safe})}{3}\lambda + 2np\ ln(p_{safe}) = 0
$$

Since $ln(p_{safe}) < 0$, the only positive solution is

$$
\lambda = -\frac{ln(p_{safe})}{3} + \sqrt{\frac{ln(p_{safe})^2}{9} - 2np\ ln(p_{safe})}
$$

■

Theorem 9 *With probability p_{safe} the number of flows passing the parallel multistage filter is bound by*

$$n_{pass} \leq b - 1 + \lfloor n\left(\frac{1}{k-1}\right)^d + -\frac{ln(p_{safe})}{3} + \sqrt{\frac{ln(p_{safe})^2}{9} - 2n\left(\frac{1}{k-1}\right)^d ln(p_{safe})}\rfloor$$

Proof We divide the flows into two groups: flows strictly above $\frac{C}{b}$ and flows below it. There are at most $b-1$ with $s > \frac{C}{b}$ and we assume that all of these pass. With lemma 1 we bound the probability of passing for flows below $\frac{C}{b}$ by $(\frac{1}{k}\frac{T}{T-C/b})^d = (\frac{1}{k-1})^d$. The number of flows in this group is at most n. By applying corollary 8.1 we can bound the number of flows from this group passing the filter. Adding the numbers for the two groups gives us exactly the bound we need to prove. ■

For our algorithm strengthening this theorem we will divide the flows above $\frac{C}{b}$ into $k-2$ groups. The first group will contain all flows of $s > T\frac{k-2}{k}$ and we will assume that all of these pass. The jth group will contain flows of sizes between $T\frac{k-j-1}{k} < s \leq T\frac{k-j}{k}$. The last ($k-1$th) group will contain as in the case above, the flows with sizes below $\frac{C}{b} = \frac{T}{k}$.

Lemma 10 *The probability of an individual flow from group j passing the filter p_j and the number of flows in group j n_j will be bound by*

$$p_j \leq \left(\frac{1}{j}\right)^d$$

$$n_j \leq \begin{cases} \lfloor\frac{b}{k-j-1}\rfloor & \textit{if } j < k-1 \\ n & \textit{for the last group} \end{cases}$$

Proof For group 1 we have $p_1 \leq 1$, so it is a correct upper bound. For all the other groups we have an upper bound on the size of flows. Using lemma 1 we see that no flow has a probability of passing larger that the probability for the largest permitted flow size. The bound for p_j is immediate.

For the last group the bound $n_{k-1} \leq n$ trivially holds because n is the total number of flows. All the other groups have a lower bound on the size of their flows. We know that the flows a group can not add up to more than the capacity of the link C. The bound on n_j is immediate. ■

Lemma 11 *If the distribution of flow sizes is Zipf, the number of flows in group j n_j will be bound by*

$$n_j \leq \begin{cases} \lfloor\frac{b}{(k-2)ln(n+1)}\rfloor & \textit{for the first group} \\ \lfloor\frac{b}{(k-j-1)ln(n+1)}\rfloor - \lfloor\frac{b}{(k-j)ln(2n+1)}\rfloor & \textit{if}(j > 1 \textit{ and } j < k-1) \\ n - \lfloor\frac{b}{(k-j)ln(2n+1)}\rfloor & \textit{for the last group} \end{cases}$$

Proof By applying lemma 12, through simple manipulations, we obtain that the number of flows i larger than $T\frac{k-j}{k}$ is bound by

$$\lfloor\frac{b}{(k-j)ln(2n+1)}\rfloor \leq i \leq \lfloor\frac{b}{(k-j)ln(n+1)}\rfloor$$

Using these bounds, the lemma is immediate. ■

We can strengthen the bound from theorem 9 by applying lemma 8.1 to these groups. Each group will have a limit on the number of passing flows. For the first group this will be the number of flows. The

```
COMPUTEBOUND(psafe)
    psafe = psafe/(k − 2)
    for j = 1 to k − 1
        p[j] = 1/(j^d)
        n[j] = COMPUTEMAXFLOWCOUNT(j)
        expectedpass[j] = n[j] ∗ p[j]
        smallest[j] = T ∗ (k − 1 − j)/k
        if (j == 1)
            worstcasepass[j] = n[j]
        else
            lambda[j] = COMPUTELAMBDA(expectedpass[j], psafe)
            worstcasepass[j] = ⌊min(expectedpass[j] + lambda[j], n[j])⌋
        endif
    endfor
    passingflows = 0
    passingtraffic = 0
    for j = k − 1 to 1
        newtraffic = worstcasepass[j] ∗ smallest[j]
        if(newtraffic + passingtraffic > C)
            worstcasepass[j] = (C − passingtraffic)/smallest[j]
            newtraffic = worstcasepass[j] ∗ smallest[j]
        endif
        passingflows+ = worstcasepass[j]
        passingtraffic+ = newtraffic
    endfor
    return passingflows
```

Figure 10: Algorithm for computing a strong high probability bound on the number of flows passing a parallel filter

probability of the total number of flows passing the filter exceeding the sum of these limits will be bound by the sum of the probabilities of individual groups exceeding their bounds. We divide p_{safe} evenly between the last $k-2$ groups.

There is one further optimization, we can apply in the distribution free case. Since we derive the limits separately for the groups, it can happen that when we add up all the passing flows, we obtain a traffic larger than C. We can discard the largest flows until the size of the passing flows is C. Figure 10 gives the pseudocode of the resulting algorithm.

B Analysis of the memory requirements of our algorithms under the assumptions that the flow sizes have a Zipf distribution

In this section we derive bounds on the number of memory entries required by sample and hold and multistage filters assuming the flow sizes have a Zipf distribution with parameter 1.

B.1 Sample and hold with a Zipf distribution of flow sizes

Lemma 12 *If the sizes of flows have a Zipf distribution, we can bound from above and below the size of the i-th flow by* $\frac{C}{i\ ln(2n+1)} \leq s_i \leq \frac{C}{i\ ln(n+1)}$.

Proof The sizes of flows are $s_i = \gamma \frac{1}{i}$. We know that $\sum_{i=1}^{n} s_i = C$.

$$\begin{array}{lll} \int_i^{i+1} \frac{1}{x}dx & \leq \frac{1}{i} & \leq \int_{i-0.5}^{i+0.5} \frac{1}{x}dx \\ \gamma \int_1^{n+1} \frac{1}{x}dx & \leq \sum_{i=1}^{n} s_i & \leq \gamma \int_{0.5}^{n+0.5} \frac{1}{x}dx \\ \gamma ln(n+1) & \leq C & \leq \gamma(ln(n+0.5) - ln(0.5)) = \gamma\ ln(2n+1) \end{array}$$

■

Corollary 12.1 *If the sizes of flows have a Zipf distribution, the number of flows above a certain threshold T is at most* $\lfloor \frac{C}{T\ ln(n+1)} \rfloor$.

Corollary 12.2 *If the sizes of flows have a Zipf distribution, the number of flows above a certain threshold T is at least* $\lfloor \frac{C}{T\ ln(2n+1)} \rfloor$.

Lemma 13 *The first x flows represent at least a fraction of* $\frac{ln(x+1)}{ln(2n+1)}$ *of the total traffic.*

Proof

$$\sum_{i=1}^{x} s_i \geq \gamma ln(x+1) \geq \frac{C}{ln(2n+1)} ln(x+1)$$

■

Based on this, we can compute that the total traffic of the first j flows is at least $C\frac{ln(j+1)}{ln(2n+1)}$. The exepected number of entries needed will be $j + Cp(1 - \frac{ln(j+1)}{ln(2n+1)})$. By differentiating, we se that we obtain the lowest value for the number of entries by choosing $j = \frac{Cp}{ln(2n+1)} - 1$.[22] By substituting we obtain the number of entries we need in the flow memory $Cp(1 - \frac{ln(Cp) - ln(ln(2n+1)) - 1}{ln(2n+1)}) - 1$. The standard deviation of the number of sampled packets belonging to flows smaller than the jth is $\sqrt{Cp(1-p)(1 - \frac{ln(j+1)}{ln(2n+1)})}$. Applying Chebyshev's inequality we obtain that the probability that the number of entries required be larger than $Cp(1 - \frac{ln(Cp) - ln(ln(2n+1)) - 1}{ln(2n+1)}) - 1 + k\sqrt{Cp(1-p)(1 - \frac{ln(j+1)}{ln(2n+1)})}$ is less than $\frac{1}{k^2}$.

[22] Actually we have to choose either the integer just below or the one just above this value, but we ignore this detail for simplicity.

B.2 Multistage filters with a Zipf distribution of flow sizes

For proving theorem 15 we first need a helper lemma.

Lemma 14 *For any* $\gamma > 0$ *and* $\gamma + 1.5 \leq i_0 < n$ *we have*

$$\sum_{i=i_0}^{n} \left(\frac{1}{1-\frac{\gamma}{i}}\right)^d < n + 1 - i_0 + d\gamma(ln(n+1) + \left(\frac{1}{1-\frac{\gamma}{i_0-0.5}}\right)^{d-1})$$

Proof

$$\begin{aligned}
\sum_{i=i_0}^{n} \left(\frac{1}{1-\frac{\gamma}{i}}\right)^d &= \sum_{i=i_0}^{n} \frac{i^d}{(i-\gamma)^d} = \sum_{j=i_0-\gamma}^{n-\gamma} \frac{(j+\gamma)^d}{j^d} = \sum_{j=i_0-\gamma}^{n-\gamma} \sum_{m=0}^{d} \frac{\binom{d}{m} j^{d-m}\gamma^m}{j^d} \\
&= \sum_{m=0}^{d} \binom{d}{m} \gamma^m \sum_{j=i_0-\gamma}^{n-\gamma} j^{-m} \leq \sum_{m=0}^{d} \binom{d}{m} (-\gamma)^m \int_{j=i_0-\gamma-0.5}^{n-\gamma+0.5} j^{-m} dj \\
&= n + 1 - i_0 + \gamma d \int_{j=i_0-\gamma-0.5}^{n-\gamma+0.5} \frac{1}{j} dj + \sum_{m=2}^{d} \binom{d}{m} \gamma^m \int_{j=i_0-\gamma-0.5}^{n-\gamma+0.5} j^{-m} dj \\
&= n + 1 - i_0 + d\gamma ln(\frac{n-\gamma+0.5}{i_0-\gamma-0.5}) + \sum_{m=2}^{d} \binom{d}{m} m\gamma^m((i_0-\gamma-0.5)^{-m+1} - (n-\gamma+0.5)^{-m+1})
\end{aligned}$$

$$\begin{aligned}
\sum_{m=2}^{d} \binom{d}{m} m\gamma^{m-1}(a^{-m+1} - b^{-m+1}) &= d\sum_{m=2}^{d} \binom{d-1}{m-1} (\left(\frac{\gamma}{a}\right)^{m-1} - \left(\frac{\gamma}{b}\right)^{m-1}) = d\sum_{r=1}^{d-1} \binom{d-1}{r} (\left(\frac{\gamma}{a}\right)^{r} - \left(\frac{\gamma}{b}\right)^{r}) \\
&= d\sum_{r=0}^{d-1} \binom{d-1}{r} \left(\frac{\gamma}{a}\right)^r - d\sum_{r=0}^{d-1} \binom{d-1}{r} \left(\frac{\gamma}{b}\right)^r \\
&= d\left(\left(1+\frac{\gamma}{a}\right)^{d-1} - \left(1+\frac{\gamma}{b}\right)^{d-1}\right) = d\left(\left(\frac{a+\gamma}{a}\right)^{d-1} - \left(\frac{b+\gamma}{b}\right)^{d-1}\right)
\end{aligned}$$

By combining these two results we immediately obtain

$$\begin{aligned}
\sum_{i=i_0}^{n} \left(\frac{1}{1-\frac{\gamma}{i}}\right)^d &\leq n + 1 - i_0 + d\gamma \left(ln\left(\frac{n-\gamma+0.5}{i_0-\gamma-0.5}\right) + \left(\frac{i_0-0.5}{i_0-\gamma-0.5}\right)^{d-1} - \left(\frac{n+0.5}{n-\gamma+0.5}\right)^{d-1}\right) \\
&< n + 1 - i_0 + d\gamma \left(ln(n+1) + \left(\frac{1}{1-\frac{\gamma}{i_0-0.5}}\right)^{d-1}\right)
\end{aligned}$$

■

Theorem 15 *If the flows sizes have a Zipf distribution, the expected number of flows passing a parallel multistage filter is bound by*

$$E[n_{pass}] \leq i_0 + \frac{n}{k^d} + \frac{db}{k^{d+1}} + \frac{db\ ln(n+1)^{d-2}}{k^2\left(k\ ln(n+1) - \frac{b}{i_0-0.5}\right)^{d-1}} \tag{5}$$

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

Proof We divide the flows into two groups. As in the general case, for the larger ones we will assume they will pass. For the smaller ones we will use lemma 14 to bound the expected value of the number of flows passing. Before deciding where to separate the two groups we will give the general formula for the second one using lemma 1 (i_0 is the rank of the largest flow in this group).

$$\begin{aligned} E[n_{small_{pass}}] &= \sum_{i=i_0}^{n} p_{s_i} \leq \sum_{i=i_0}^{n} \left(\frac{1}{k}\frac{T}{T-s_i}\right)^d \leq \frac{1}{k^d}\sum_{i=i_0}^{n}\left(\frac{T}{T-\frac{C}{i\ ln(n+1)}}\right)^d \\ &= \frac{1}{k^d}\sum_{i=i_0}^{n}\left(\frac{1}{1-\frac{\frac{b}{k\ ln(n+1)}}{i}}\right)^d \end{aligned}$$

For lemma 14 to apply we need $i_0 \geq 1.5 + \frac{b}{k\ ln(n+1)}$. To be able to bound the probability of these flows passing the filter, by lemma 4 we need $s_{i_0} \leq T\frac{k-1}{k}$. Through lemma 12 we obtain $i_0 \geq \frac{\gamma k}{T(k-1)} \geq \frac{b}{ln(2n+1)(k-1)}$. To satisfy both inequalities we set i_0 to $\lceil max(1.5 + \frac{b}{k\ ln(n+1)}, \frac{b}{ln(2n+1)(k-1)})\rceil$.

$$\begin{aligned} E[n_{pass}] &= \sum_{i=1}^{n} p_{s_i} = \sum_{i=1}^{i_0-1} p_{s_i} + \sum_{i=i_0}^{n} p_{s_i} \leq i_0 + \frac{n+1-i_0+\frac{db}{k\ ln(n+1)}\left(ln(n+1)+\frac{1}{\left(1-\frac{b}{k\ ln(n+1)(i_0-0.5)}\right)^{d-1}}\right)}{k^d} \\ &\leq i_0 + \frac{n}{k^d} + \frac{db}{k^{d+1}} + \frac{db\ ln(n+1)^{d-2}}{k^2\left(k\ ln(n+1)-\frac{b}{i_0-0.5}\right)^{d-1}} \end{aligned}$$

■

C Defining large flows with leaky buckets

In this appendinx we propose an alternate definition of large flows based on leaky buckets instead of measurement intervals. We also show how to adapt the multistage filters to this new definition and provide an analytical evaluation of the new scheme.

Defining large flows based on measurement intervals can lead to some unfairness. For example if a flow sends a burst of size slighlty larger than the threshold T within one measurement interval it is considered large. However, if the same burst spans two intervals it's not. Even flows sending bursts of size almost $2T$ are not considered large if the bursts span measurement intervals a certain way. It can be argued that we should consider to be a large flow all flows that send more than T over any time interval no longer than a measurement interval. While this distinction is arguably not very important for the case of traffic measurement, it might matter for other applications.

We use a leaky bucket descriptor (also known as linearly bounded arrival process) to define large flows: a flow is large if during any time interval of size t it sends more than $r*t+u$ bytes of traffic. By properly choosing the parameters of the leaky bucket descriptor, we can ensure that all flows that send T bytes of traffic over a time interval no longer than a measurement interval are identified. We can adapt the multistage filters to this new definition by replacing the counters with "leaky buckets" and instead of looking for counters above the threshold we look for buckets that violated the descriptor. We will first discuss how we can implement these efficiently at high speeds, and then give an analytical evaluation of the new algorithm.

C.1 Analytical evaluation of the parallel multistage filter using leaky buckets

Flows sending more than $r * t + u$ in any time interval of length t are large. For the example we used in section 4.2 by setting r to 0.5 Mbytes/s and u to 0.5 Mbytes, we are guaranteed that flows that send 1 Mbyte during any second are labeled as large. This guarantees that we catch all flows that send more than 1 Mbyte during a measurement interval. We can conceptually describe the operation of the buckets as follows. Each bucket has a counter c initialized to 0. Every $\frac{1}{r}$ seconds this counter is decremented by 1 unless it is already 0 . When a packet of size s arrives, its size is added to the counter, but the value of the counter is not incresed above u. If the counter is u the incoming packet is considered to belong to a large flow. We also use the phrases the bucket is in violation and the packet passes the bucket to describe this situation. Section C.2 desctribes how this can be implemented efficiently. Actual implementations would probably use an approximation of this algorithm (e.g. they might decrement the leaky bucket less often), but we are not concerned with these details in our analysis. We use the notations below in our analysis.

- r the steady state data rate of the leaky bucket;
- u the burst size of the leaky bucket;
- C the data rate of the link (in bytes/second);
- k the stage strength: the ratio of r average data rate of the traffic through a bucket $k = \frac{r\ b}{C}$ (in our modified example above k is 5);
- τ the drain time for the leaky bucket $\tau = \frac{u}{r}$, for our example $\tau = 1$ second;
- c the counter of a certain leaky bucket (see below);
- a the number of "active" buckets in a stage (buckets with non-zero counters);
- A the active traffic in a particular stage defined as the sum of all counters;
- s the size of a packet or a sequence of packets;

We formalize the description of how the leaky buckets of the stages operate in the following two lemmas.

Lemma 16 *If $c_{initial}$ is the initial value of the counter of a bucket, after a time t where the bucket received no packets the value of the counter will be $c_{final} = max(0, c_{initial} - rt)$.*

Lemma 17 *If $c_{initial}$ is the initial value of the counter of a bucket when it receives a packet of size s, the value after the packet was processed is going to be $c_{final} = min(c_{initial} + s, u)$.*

Now we can prove a lemma that will help use prove we have no false negatives.

Lemma 18 *Let c be the value of the counter of a leaky bucket tracking a flow. Let c' be the counter of another bucket that counts all the packets of our flow and possibly packets of other flows. For any moment in time $c \leq c'$*

Proof By induction on time using as steps the moments when the packets are received.

Base case The buckets are exactly identical at the beginning of the interval $c = c`$.

Inductive step Three things can happen: a packet belonging to the flow arrives, a packet not belonging to the flow arrives or no packets arrive for time t. In all three cases we will use the fact that by induction hypothesis, $c \leq c'$ in the beginning. If a packet of size s belonging to the flow arrives, by lemma 17 we have $c_{new} = min(c + s, u)$ and $c'_{new} = min(c' + s, u)$ therefore $c_{new} \leq c'_{new}$. If a packet of size s not belonging to the flow arrives, by lemma 17 we have $c_{new} = c$ and $c'_{new} = min(c' + s, u)$ therefore $c_{new} \leq c'_{new}$. If no packets arrive for time t, by lemma 16 at the end of the interval we have $c_{new} = max(0, c - rt)$ and $c'_{new} = max(0, c' - rt)$, therefore $c_{new} \leq c'_{new}$. ■

Corollary 18.1 *Let t be the moment in time when a certain flow exceeds the leaky bucket descriptor. The violation will be detected by the leaky bucket at time t no matter how many packets belonging to other flows hash to the same bucket.*

Theorem 19 *A parallel multistage filter will detect any flow exceeding the leaky bucket descriptor at latest when it does so.*

Proof By corollary 18.1, at all stages, the buckets the flow hashes to will detect the leaky bucket descriptor violation for the first packet of the flow that violates it, therefore this packet will pass the filter causing the flow to be detected. ■

Just as in the case with the measurement intervals, we can have no false negatives and we want to bound the number of false positives. What we want to bound is the number of flows passing the filter during a certain time interval which gives us the peak rate at which new flows are added to the flow memory.

Lemma 20 *For any time interval t, if the counter of a bucket at the beginning was $c_{initial}$ and the traffic that hit the bucket during the interval is s, the final value of the counter is bound by $c_{final} \leq max(0, c_{initial} - rt) + s$.*

Proof By induction on time, using as steps the moments when the packets are received.

Base case At the beginning of the experiment, the time passed since the beginning of the experiment will be $t = 0$ and the sum of the sizes of the packets sent will be $s = 0$ therefore $c = c_{initial} = max(0, c_{initial} - rt) + s$.

Inductive step Two things can happen: a packet arrives or no packets arrive for time t'. In all cases we will use the fact that by induction hypothesis, $c \leq max(0, c_{initial} - rt) + s$ in the beginning where t is the time that passed since the beginning of the experiment and s is the sum of the sizes of the packets received so far. If a packet of size s' arrives, by lemma 17 we have $c_{new} = max(c + s', u) \leq c + s' \leq max(0, c_{initial} - rt) + s + s'$. If no packets arrive for time t', by lemma 16 at the end of the interval we have $c_{new} = max(0, c - rt') \leq max(0, max(0, c_{initial} - rt) + s - rt') \leq max(0, max(0, c_{initial} - rt) - rt') + s \leq max(0, max(0, c_{initial} - rt - rt')) + s = max(0, c_{initial} - r(t + t')) + s$. ■

Corollary 20.1 *The value of the counter of the bucket is not larger than the amount of traffic that bucket received during the last* τ*.*

Lemma 21 *The active traffic in any stage is bound by* $A \leq \frac{bu}{k}$*.*

Proof By corollary 20.1, the size of each individual bucket will be bound by the traffic it received during the last τ, therefore A will be bound by the total traffic received during this interval which is bound by $C\tau = \frac{bu}{k}$. ■

Corollary 21.1 *At any moment in time the number of buckets in a stage* a_x *with* $c \geq x$ *is bound by* $a_x \leq \lfloor \frac{bu}{kx} \rfloor$*.*

We will bound the expected number of flows passing the filter during an interval of τ, the drain time for the leaky bucket. We cannot directly use corollary 21.1 because a particular flow might pass the filter at any moment during the interval of τ.

Lemma 22 *The number of buckets of a stage with* $c \geq x$ *at any time during an interval of* τ *is bound by* $a_x \leq \lfloor 2\frac{bu}{kx} \rfloor$*.*

Proof By lemma 21.1, the maximum number of buckets above x at the start of the interval is $\lfloor \frac{bu}{kx} \rfloor$ with the rest of the active traffic in other buckets. The best way an adversary could use the remaining active traffic at the beginning at the interval and the traffic sent during the interval is to fill buckets one by one. Since the amount of traffic sent during the interval is bound by $\frac{bu}{kx}$, by adding the number of buckets that were above c at the beginning to the ones that got filled up during the interval we obtain the bound of this lemma.■

Lemma 23 *For a flow that sends a total of* s *bytes during an interval* τ *and the preceding* τ *seconds, the probability that any of its packets pass the parallel multistage filter during the interval is bound by* $p_s \leq \left(\frac{2}{k}\frac{u}{u-s}\right)^d$*. If* $s \leq u\frac{k-2}{k}$ *this bound is below 1.*

Proof By lemma 20.1, the size of the buckets is hashes to is bound by the traffic they received in the past τ seconds. This traffic is made up by traffic of the flow we are analyzing and traffic of other flows $c \leq s + s_{rest}$. The amount of traffic our flow sends during any window of τ seconds ending in the interval is bound by s. For the flow to pass the filter, we need all buckets to pass the flow $s + s_{rest} \geq u$. By an argument similar to the one on lemma 22, the number of bucket at each stage for which $s_{rest} \geq u - s$ at any moment during the interval is bound by $a_{u-s} \leq 2\frac{bu}{k(u-s)}$. Therefore the probability of passing any single stage is bound by $\frac{2u}{k(u-s)}$. This gives us the bound on the probability for a flow passing all of the stages as in the lemma. ■

Notice that this lemma is an upper bound, not the actual probability. It is even further from the real probability for the flow passing the filter than lemma 1 because it assumes that for all stages s_{rest} reaches the right value exactly when the last packet of the flow is sent. This is quite unlikely in practice. Based on this lemma, we can give our final bound for the expected number of flows passing the filter during the interval τ.

Theorem 24 *The expected number of flows passing a multistage parallel filter during any interval of length* τ *is bound by*

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

Proof Let s_i be the sequence of flow sizes present in the traffic mix counting the traffic sent during the interval and the τ preceding seconds. Let n_i the number of flows of size s_i. $h_i = \frac{n_i s_i}{2C\tau}$ is the share of the total traffic the flows of size s_i are responsible for. We have $\sum n_i = n$ (n is defined as the number of flows active during the interval, not the interval and the τ preceding second), and $\sum h_i = 1$. By lemma 23 the expected number of flows of size s_i to pass the filter is $E[n_{i_{pass}}] = n_i p_{s_i} \leq$. 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 $s_i > u\frac{k-2}{k}$. For these $E[n_{i_{pass}}] \leq n_i = \frac{h_i 2C\tau}{s_i} < \frac{h_i 2C\tau}{u\frac{k-2}{k}}$, therefore substituting them with a number of flows of size $u\frac{k-2}{k}$ that generate the same amount of traffic is guaranteed to not decrease the lower bound for $E[n_{pass}]$. The smallest flows are the ones below the average flow size of $\frac{2C\tau}{n}$. For these $p_{s_i} \leq p_{\frac{2C\tau}{n}}$. The number of below average flows is bound by n. For all these flows taken together $E[n_{small_{pass}}] \leq np_{\frac{2C\tau}{n}}$.

$$
\begin{aligned}
E[n_{pass}] &= \sum E[n_{i_{pass}}] = \sum_{s_i > u\frac{k-2}{k}} E[n_{i_{pass}}] + \sum_{\frac{2C\tau}{n} \leq s_i \leq u\frac{k-2}{k}} E[n_{i_{pass}}] + \sum_{s_i < \frac{2C\tau}{n}} E[n_{i_{pass}}] \\
&\leq \sum_{s_i > u\frac{k-2}{k}} \frac{h_i 2C\tau}{s_i} + \sum_{\frac{2C\tau}{n} \leq s_i \leq u\frac{k-2}{k}} \frac{h_i 2C\tau}{s_i} \left(\frac{2}{k}\frac{u}{u-s_i}\right)^d + n\left(\frac{2}{k}\frac{u}{u-\frac{2C\tau}{n}}\right)^d \\
&\leq 2C\tau \left(\sum_{s_i > u\frac{k-2}{k}} h_i \frac{1}{u\frac{k-2}{k}} + \sum_{\frac{2C\tau}{n} \leq s_i \leq u\frac{k-2}{k}} h_i \frac{1}{s_i}\left(\frac{2}{k}\frac{u}{u-s_i}\right)^d \right) + n\left(\frac{2}{k}\frac{nu}{nu-2C\tau}\right)^d \\
&\leq \frac{2ub}{k} \max_{\frac{2C\tau}{n} \leq s_i \leq u\frac{k-2}{k}} \left(\frac{1}{s_i}\left(\frac{2}{k}\frac{u}{u-s_i}\right)^d\right) + n\left(\frac{2n}{kn-2b}\right)^d
\end{aligned}
$$

As we saw in the proof of theorem 3, the maximum is reached at one of the ends of the interval. By substituting these values we obtain the bound. ■

If we compute the number for our example we obtain a bound of 5,202.7 flows which is much higher than the 121.2 theorem 3 gave. But is the comparison fair? Are the problems solved in the two cases equivalent? In te analysis with measurement intervals the number of flows that could violate the threshold during the measurement interval is 100. What is this number in our case? We can have 199 flows that keep their buckets at 0.5 Mbytes before our interval starts and they send one single small packet during the interval. These packets are all in violation and they should be detected. After this, we can have 198 other flows sending bursts of slightly more than 0.5 Mbytes so that they violate their leaky bucket descriptor. These flows should also all be passed by the filter if it is to avoid false negatives. Therefore we have a traffic pattern that requires at least 397 flows to be detected during the interval. If we proportionately increase the number of buckets at each stage from 1000 to $b = 4000$, theorem 24 gives us a bound of 454.6 which is approximately 4 times the bound of theorem 3. As with that result, we expect that in practice the number of flows passing will be much smaller.

C.2 Implementing multistage filters with leaky buckets

A naive implementation of the leaky buckets that make up the stages would keep decrementing the counters by 1 every $1/r$ seconds. This needs a lot of memory accesses and is not necessary. We think of the counters as numbers that move between 0 and u and what matters to the algorithm is where the counters are within this interval. Instead of decrementing all the counters every $1/r$ seconds by one, we can move the interval: we will have a virtual 0 and a virtual u that get *incremented* every $1/r$ seconds. Since we can keep these values in two registers, incrementing them often does not pose problems. With these new definitions, the counters themselves work the following way: when a new packet hashes to the counter, we first check if the value if the counter is below the virtual 0 we update it to 0; we add the size of the packet to the counter and if it is above the virtual u, we decrement it to virtual u; finally if the counter reached the virtual u we declare that the bucket is in violation. While this might sound long, it needs no more memory accesses than the counters of filters operating with measurement intervals. With this implementation, we need to worry about overflows. We can implement the operations in such a way that when the virtual 0 and virtual u overflow, comparisons and arithmetic operations still work correctly. However, after an overflow an old counter that received no packets can seem to have a very large value instead of a very small one. To solve this problem we can use a background process that periodically updates to virtual 0 all the counters below it. Improvements to the basic parallel filter such as shielding and conservative update easily generalize to our filter using leaky buckets.

D Heuristic rules for tight configuration of the multistage filters

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.

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 6 gives the optimal value for b and Equation 7 gives the total amount of memory required with this choice of b.

$$b = \Delta l \sqrt[d+1]{\frac{2n}{r\Delta l}} \tag{6}$$

$$m_{total} = l + \Delta l + (d+1) r \Delta l \sqrt[d+1]{\frac{2n}{r\Delta l}} \tag{7}$$

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 7 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.

E Cisco NetFlow

NetFlow [17] is a feature of Cisco routers that implements per flow traffic measurement. It is one of the primary tools used to collect traffic data by large transit ISPs today [9]. NetFlow is intended (by Cisco) to serve as a basis for usage based billing. We briefly discuss here some details of Cisco NetFlow. We also present an analytical evaluation of the accuracy of sampled NetFlow and its memory requirements. At the end of this appendix we propose an alternative implementation solution that could increase by an order of magnitude the link speeds NetFlow can handle without resorting to sampling. This implementation procedure can also be used in conjunction with our algorithms.

E.1 Basic NetFlow

NetFlow defines flows as unidirectional streams of packets between two particular endpoints. A flow is identified by the following fields: source IP address, destination IP address, the protocol field in the IP header, source port, destination port, the TOS byte and the interface of the router that received the packet. In the DRAM of the router interface card there is a flow cache that stores per flow information (we call it flow memory in this paper). The entry for a flow holds, besides the flow identifier, various types of information about the flow: timestamp of when the flow started and ended, packet count, byte count, TCP flags, source network, source AS (Autonomous System), destination network, destination AS, output interface, next hop router. Various heuristics (e.g. flows that have been inactive for a particular period of time, the RST and FIN TCP flags) are used to determine when a flow ends.

The NetFlow data captured by at the router is exported via UDP packets to computers that process it further. The raw NetFlow data can be processed in a variety of ways and can give all kinds of information about the traffic. There are two major problems with the basic NetFlow: for interfaces faster than OC3 updating the flow cache slows down the operation of the interface and the amount of data generated by NetFlow can be so large that it overwhelms the collection server or its network connection ([9] reports loss rates of up to 90%). Cisco's solution to the first problem is sampling packets and to the second aggregating the measurement data on the router.

E.2 NetFlow Aggregation

Many applications are not interested in the raw NetFlow data, but in an aggregated form of it. For example when deriving traffic demands one is interested by traffic between networks (more exactly IP prefixes), not individual endpoints: all NetFlow records of individual flows whose two endpoints are in the same two networks are aggregated together. One can also imagine arrangements between ISPs with payment based on traffic that would require a similar type of aggregation.

Cisco's solution to the problem of NetFlow generating too much data was introduced in IOS 12.0(3)T . The aggregation of raw data is performed at the router. One or more extra caches called aggregation caches are maintained at the router. Only the aggregate data is exported thereby reducing substantially the amount of traffic generated. Five aggregation schemes are currently supported: based on source and destination AS, based on destination prefix, based on source prefix, based on source and destination prefix and based on source and destination ports.

E.3 Sampled NetFlow

Cisco introduced a feature called sampled NetFlow [18] with high end routers. The performance penalty of updating the flow cache from DRAM is avoided by sampling the traffic. For a configurable value of a parameter x, one of every x packets is sampled. The flow cache is updated only for the sampled packets. Even though the update operation is not performed any faster, since it is performed less often it does not affect the performance of the router. Cisco recommends that sampling is turned on for interfaces above OC-3. The advantage of this solution is that it is very simple and requires no significant changes to the hardware of the line card.

E.4 The accuracy of sampled NetFlow

The actual sampled NetFlow works by counting every x-th packet irrespective of packet sizes. To simplify the analysis we will assume that all packets have the same size y and are sampled with probability $p = 1/x$.

Let c be the number of packets counted for a given flow and s the actual size of the flow (in packets). The probability distribution of c is binomial. The probability that a flow of size s is missed is the same as the probability that no packets get sampled which is $(1-p)^s$. By the linearity of expectation we obtain that $E[c] = sp$. Therefore the best estimate for s is c/p. Since the probability distribution for c is binomial, its standard deviation will be $SD[c] = \sqrt{sp(1-p)}$. The standard deviation of our estimate of s will be $1/p\sqrt{sp(1-p)}$.

To compare the accuracy of sampled NetFlow with our algorithms we compute the standard deviation of the estimate of the size of the flow that is at the threshold $T = s * y$ (in bytes). By substituting in the formula above, this is $y/p\sqrt{p(1-p)T/y} = \sqrt{y(1-p)T/p}$. Based on this number we can also compute the relative error of a flow of size T which is $\sqrt{y(1-p)/Tp}$. We can substitute actual numbers into this formula. Since sampling is recommended above OC-3 (155.52 Mbits/s=19,440,000 bytes/s), if the line speed is x times OC-3, then the sampling probability is at most $p = 1/x$. Smaller sampling probabilities can be used to reduce the memory requirements at the cost of accuracy. Let the measurement interval be i seconds. Assuming a threshold of $T = zC = xiz19,440,000$ and a packet size of 1500 bytes (which is common for large flows), the relative error of the estimate of a flow at the threshold is $\sqrt{1500(1-1/x)x/T} \approx \sqrt{1,500/(19,440,000iz)} = 0.0087841/\sqrt{zi}$.

E.5 The memory requirements of sampled NetFlow

To be able to compare NetFlow to our algorithms, for the purpose of thia analysis 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 number of entries used by NetFlow is bound by both the maximum number of packets sampled during a measurement interval and the number of active flows n. Assuming the link is fully utilized with minimum size packets of 40 bytes, the number of packets sampled in i seconds is exactly $ipC/40$. As we saw in section E.4, the maximum sampling that doesn't slow down the packet forwarding is $p = 19,440,00/C$. If we use this sampling rate, the maximum number of updates per measurement interval is $i19,440,000/CC/40 = 486,000i$.

E.6 Keeping a queue of packet headers

The improvement presented in this section significantly increases the amount of time NetFlow can spend with each packet. It involves addition of a simple SRAM buffer.

In [13] Lakshman and Stiliadis argue that packet forwarding and classification decisions have to be made at line speed even for the smallest of packets. We argue that this does not extend to traffic measurement. We can keep the packet headers and other relevant information in a small queue and process that information (for traffic measurement purposes) at somewhat lower speeds after the packet was sent on the wire. This does not cause any delay for the actual packet. We are basically decoupling the forwarding of packets from the traffic measurement device. We argue that the benefits far outweigh the costs of this improvement.

Practically all of the packets from the traces we used are at least 40 bytes large. However the average size is around 550 bytes. If we were to dimension the traffic measurement device to handle at line speeds packets of 240 bytes instead of 40 bytes, this would give us 6 times as much time to process each packet. Since the average time the traffic measurement device has to process a packet is more than twice what it needs, the SRAM buffer holding the queue of packet headers need not be very large to make it very unlikely that it ever overflows. This is very similar to how packet headers are buffered on cards used for traffic capture until the driver can handle them.

F Choosing a suitable measurement interval

In this appendix we choose the size of the measurement interval based on the traces we have. The optimal size for the measurement interval depends on both the application for our algorithms and the traffic mix. The purpose of the measurements from this appendix is not to derive a size for the measurement interval that we recommend for all applications. We only want to derive a size for the measurement interval that is close enough to what applications would use to make the results from section 7 relevant.

The task of choosing an appropriate measurement interval is further complicated by the lack of objective criteria for deciding what a good value is. If the measurement intervals are too large the data collected might be too coarse for the purposes of the application. If the interval is too small than flows that have gaps between some of their packets larger than the measurement interval can appear as repeatedly going inactive and starting to send again. This might be undesirable for the application and it can reduce the effectiveness of optimizations to our algorithms that rely on the persistence of the flows (such as preserving entries in the flow memory across measurement intervals).

What do we measure in order to determine what a good value for the measurement interval is? One would want as many as possible of the flows to send their packets spaced apart by less than the size of the measurement interval. An obvious measure of how good a size for the measurement interval is is the

Interval	MAG	IND	COS
0.1 s	4.482% /1.617% /67.261%	5.899% /7.068% /81.572%	9.923% /4.101% /77.623%
0.2 s	13.801%/7.829% /78.805%	8.809% /19.935%/87.162%	15.415%/10.481%/86.326%
0.5 s	35.556%/31.206%/91.939%	16.471%/44.679%/93.601%	23.659%/29.416%/93.629%
1.0 s	49.682%/45.012%/95.362%	27.896%/58.222%/96.651%	36.707%/48.031%/96.614%
2.0 s	56.683%/58.119%/97.224%	32.022%/67.509%/97.979%	41.659%/61.148%/97.850%
5.0 s	67.685%/76.528%/98.969%	57.919%/83.102%/99.250%	51.282%/80.745%/99.043%
10.0 s	90.056%/87.086%/99.611%	79.765%/91.705%/99.723%	63.092%/86.705%/99.483%

Table 8: Comparing measurement intervals for flows defined by 5-tuples

Interval	MAG	IND	COS
0.1 s	1.085% / 4.481%/75.160%	1.419% /13.136%/86.458%	2.373% /18.597%/89.663%
0.2 s	2.906% /10.246%/85.209%	2.884% /27.138%/91.339%	3.889% /34.595%/94.541%
0.5 s	9.683% /23.896%/95.373%	6.178% /51.617%/96.293%	6.262% /50.827%/97.919%
1.0 s	16.660%/33.579%/97.728%	11.195%/65.484%/97.871%	10.943%/60.578%/99.081%
2.0 s	21.377%/43.254%/98.780%	14.309%/73.635%/98.739%	15.162%/70.080%/99.535%
5.0 s	32.745%/59.495%/99.579%	49.080%/86.646%/99.493%	38.860%/82.997%/99.856%
10.0 s	71.205%/72.380%/99.854%	76.436%/92.668%/99.829%	61.964%/89.363%/99.941%

Table 9: Comparing measurement intervals for flows defined by destination IP

Interval	MAG
0.1 s	2.260% / 60.499% / 95.969%
0.2 s	3.975% / 73.242% / 98.031%
0.5 s	9.003% / 82.135% / 99.408%
1.0 s	14.522% / 87.148% / 99.727%
2.0 s	19.154% / 89.814% / 99.857%
5.0 s	29.707% / 94.430% / 99.947%
10.0 s	54.700% / 96.999% / 99.979%

Table 10: Comparing measurement intervals for flows defined by the pair of ASes

percentage of flows that send all their packet closer than the size of the measurement interval. But often there are many small flows that send their packets far apart while large flows send them closer. We obtained a less biased measure if we weigh the flows by the total traffic they send. While this is a good measure if flows are defined at the granularity of a TCP connection it is not that good if we look at coarser aggregates such as all packets sent to a given IP address. The reason is that there might be multiple distinct connections with packets close to each other, but spaced far apart. Even though most of the packets of such an aggregate are close some are far and it would be classified as a flow that has packets further apart than the measurement interval. We introduce the third measure as the percentage of packets (weighted by packet sizes) that arrived within a measurement interval of the previous packet of the same flow.

Table 8 shows our results for flows defined at the granularity of TCP connections by source and destination IP address and port and by protocol number, Table 9 shows our results for flows defined by the destination IP address and Table 10 shows our results for flows defined by the source and destination autonomous system. The first two tables show the results of measurements on the traces MAG, COS and IND and the third one only on trace MAG (because the other two traces are anonymized and we cannot perform route lookups on them). The values in the cells of the tables represent the 3 measures we discussed: the percentage of flows that have all their packets closer than the given interval, the same percentage weighted by the total amount of traffic transferred by the flows and the percentage of packets weighted by their size that arrived within the interval of the previous packet of the same flow. We can see that for all granularities and for all traces, a measurement interval of 5 seconds assures that 99% or more of the packets (weighted by their size) arrive within a measurement interval of the previous packet of the same flow. Based on these results we will use a measurement interval of 5 seconds in all our experiments.

G Measuring sample and hold

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. We measure the effect of early removal and determine a good value for the early removal threshold. We conclude by summarizing our findings about the sample and hold algorithm. We have 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.

G.1 Comparing the behavior of the base algorithm to the analytic results

We first look 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 (corresponding to a probability of between 37% and less than 0.1% of missing a flow at the threshold (see Section 4.1.1)). 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 11 shows the percentage of false negatives (large flows not identified). We also plot the probability of false negatives predicted by our conservative analysis (the Y axis is logarithmic). 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. The measurements confirm that the probability of false negatives decreases exponentially as the oversampling increases. Figure 12 shows the average error in the estimate of the size of an identified large flow. We also plot the analytic estimate for the difference between the estimate and the actual traffic of a large flow from Section 4.1.1. The measured error is slightly below the error predicted by the analysis. The explanation is that the analysis assumed that the size of the error is unbounded. In practice, the size of the error is bounded by the size of the flow. The measurements confirm that the average error of the estimates is proportional to the inverse of the oversampling. Figure 13 shows the maximum over the 900 measurement intervals for the number of entries of flow memory used. The measurement results are more than an order of magnitude lower than the bound from Section 4.1.2. There are two main reasons. The most obvious one is that the links are lightly loaded (between 13% and 27%) so the number of packets sampled is much smaller than for a congested link as

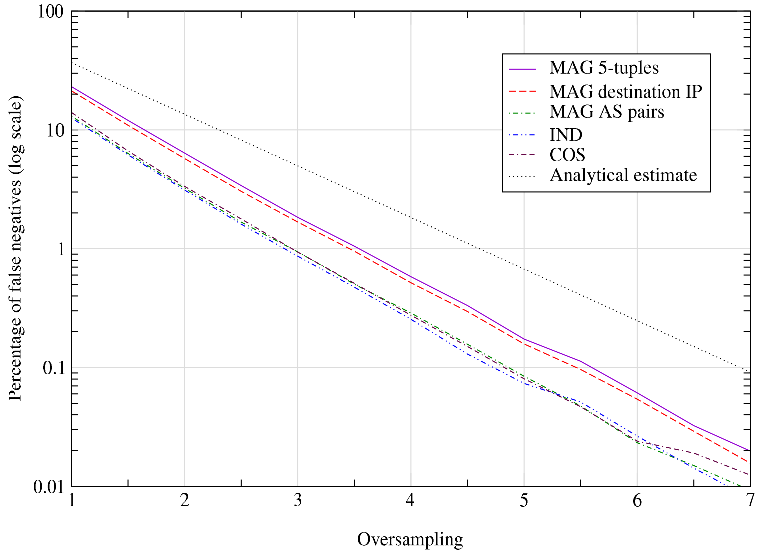

Figure 11: Percentage of false negatives as the oversampling changes

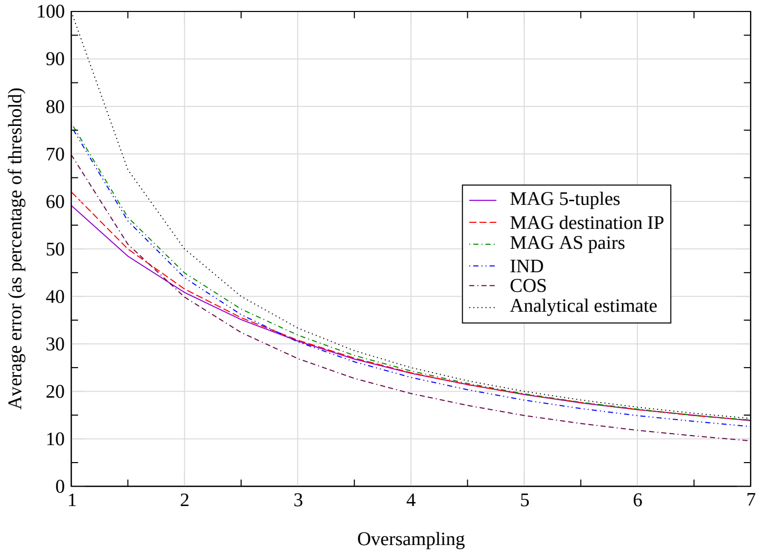

Figure 12: Average error in the traffic estimates for large flows

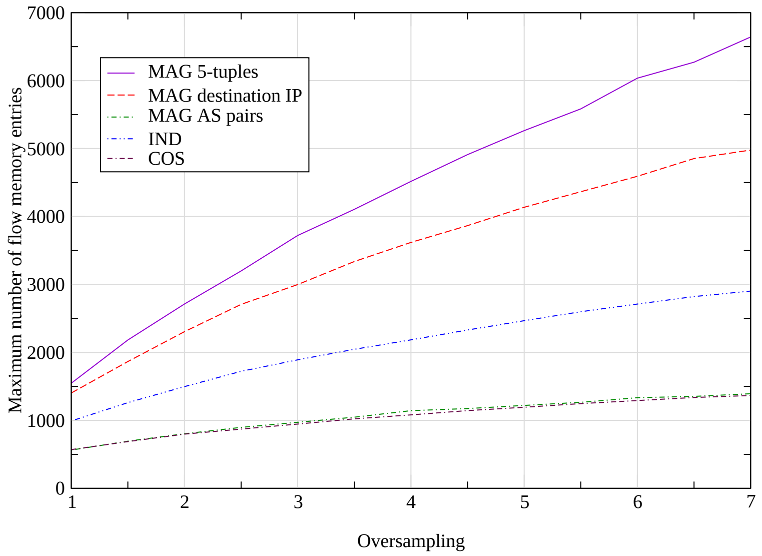

Figure 13: Maximum number of flow memory entries used

assumed by the bound. The other reason is that many of the sampled packets do not create new entries in the flow memory. This explains why the number of entries increases sub-linearly with the oversampling and not roughly linearly as predicted by the analysis. 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 look at how the choice of the threshold influences the performance of the sample and hold algorithm. We run the algorithm with a fixed oversampling of 5 for thresholds between 0.005% and 0.1% of the link bandwidth. Figure 14 shows the percentage of false negatives. As in the previous case, the actual percentage is on average between 3 and 8 times lower than the one predicted by the analysis (depending on the trace and the definition of the flow ID). The only value that is suspiciously high is the one for the MAG trace with a flow definition at the TCP granularity. Upon closer analysis of the trace we found out that there are only 3 flows (all 3 netnews transfers between the same two hosts but on different ports) that are above the threshold in all intervals and they are within 15% of the threshold. This explains why in this case the observed rate of false negatives so closely matches the prediction of the analysis. Figure 15 shows 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 to 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. This results in smaller error as illustrated by our results. Figure 16 shows the maximum number of entries of flow memory used. As before the actual number is much smaller

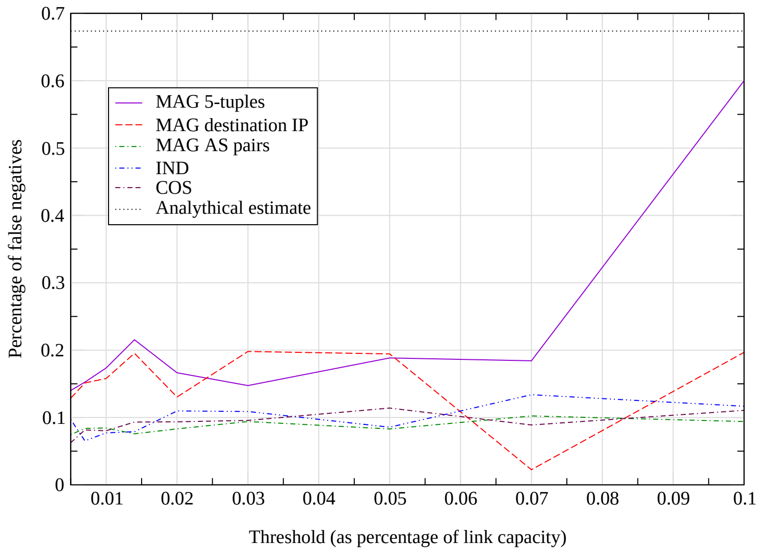

Figure 14: Percentage of false negatives as the threshold changes

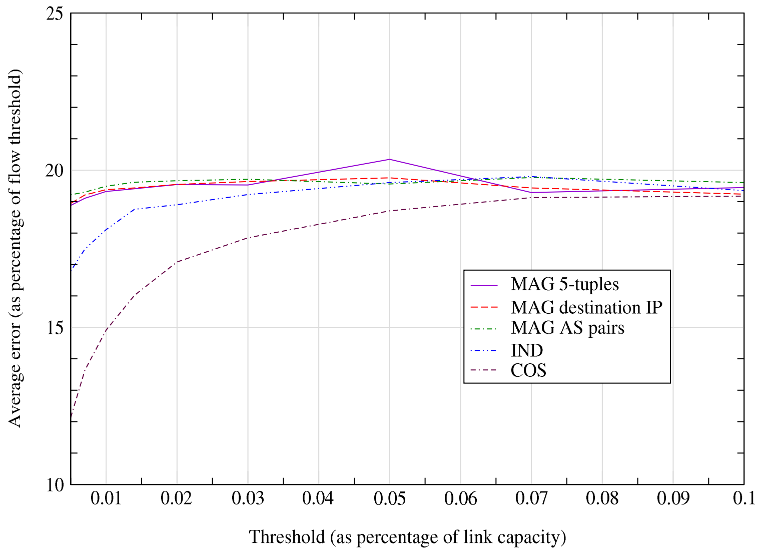

Figure 15: Average error in the traffic estimates for large flows

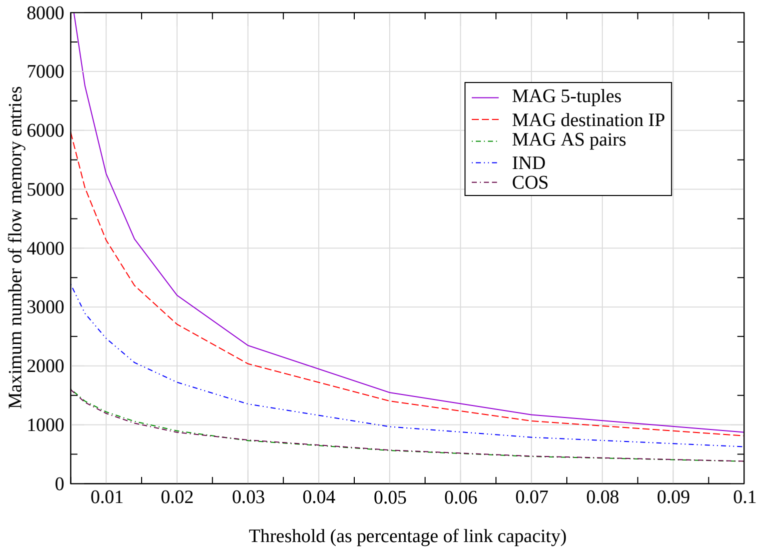

Figure 16: Maximum number of flow memory entries used

than the bound from Section 4.1.2. As the threshold decreases, the number of entries increases much faster for the traces with many flows than for the ones with few.

Findings: Sample and hold performs better than predicted by our conservative analysis. The percentage of false negatives is roughly one order of magnitude smaller than predicted in section 4.1.1 because most large flows are considerably above the threshold. The average error of the estimates is slightly below the expected value. When the threshold is the same order of magnitude as the size of the packets, the improvement is stronger. The memory requirement of the algorithm can be orders of magnitude below what Section 4.1.2 predicts. The main reasons: links are lightly loaded and large flows are sampled repeatedly.

G.2 The effect of preserving entries

In this section we measure the improvement introduced by preserving entries from one measurement interval to the next one. We compare the results with the ones from the measurements of the base algorithm. For the false negatives and average error we omit from the computation the first measurement interval because there no entries are preserved from the previous interval, making the behaviour of the algorithm identical to the original sample and hold. We perform 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. Figures 17 to 19 show the the number of false negatives, the average error of the estimate and the memory usage with preserving entries. All the plots present ratios to the values obtained without preserving

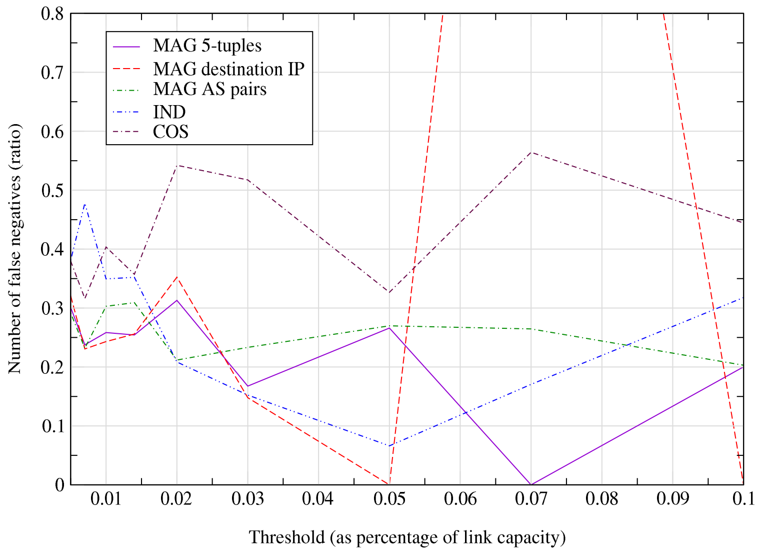

Figure 17: Effect of preserving entries on false negatives

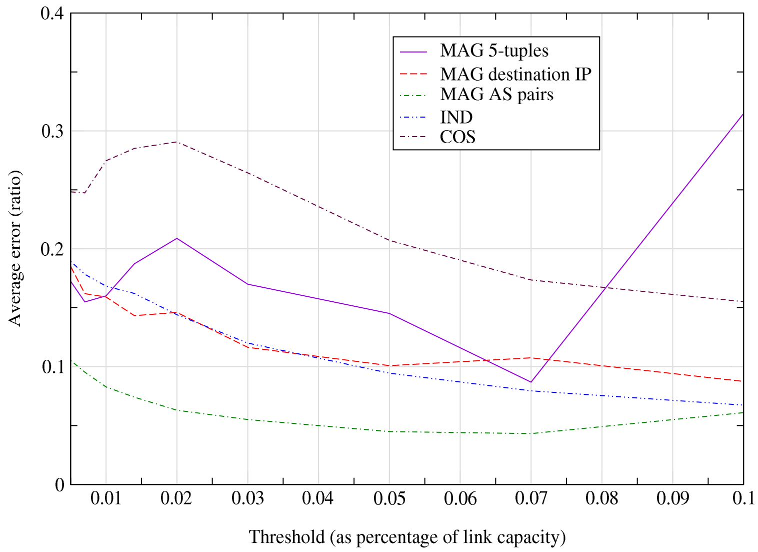

Figure 18: Effect of preserving entries on average error

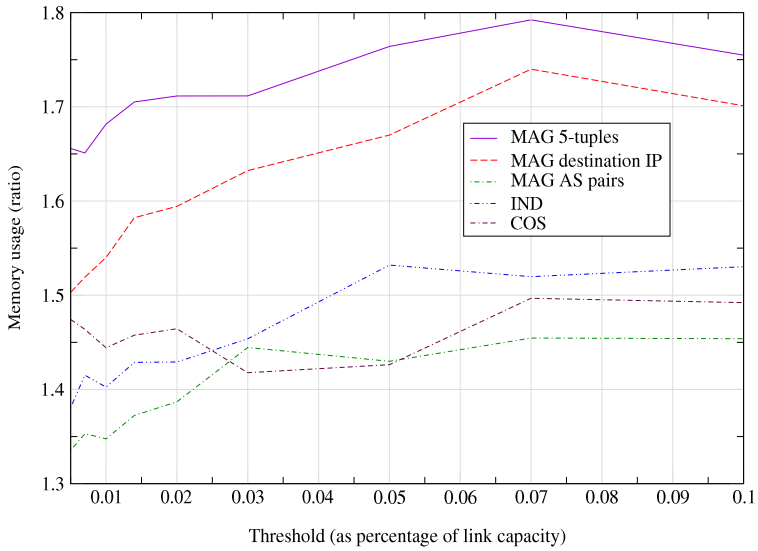

Figure 19: Effect of preserving entries on memory usage

entries. As shown in Figure 17 the number of false negatives is generally reduced to between 15% and 50% . The exact amount of the improvement depends strongly on the actual trace, the flow definition and the threshold. The huge spike for the MAG trace with flows defined based on destination IP address for a threshold of 0.07% of the link bandwidth is due to the fact that the original algorithm has a single false positive in 900 intervals while when preserving entries we have 2. We don't consider this an indication that preserving entries can increase the number of false negatives. The average error decreases to between 30% and 5% strongly depending on the trace and flow definition. We consider this the most important gain of preserving entries. The increase in memory usage is between 30% and 80% and depends strongly on the trace and flow definition. We can see that traces dominated by few very heavy very long lived flows such as MAG with flows defined by AS pairs have both a low cost (small increase in memory) and a high benefit (large decrease in error) for preserving entries. For the COS trace where few very heavy but not very long lived flows dominate, the cost of preserving entries is still low but the benefits are not as high.

Findings: Preserving entries reduces 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. The value of the oversampling does not affect the magnitude of the improvements of preserving entries.

G.3 The effect of early removal

To measure the effect of early removal, we choose 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

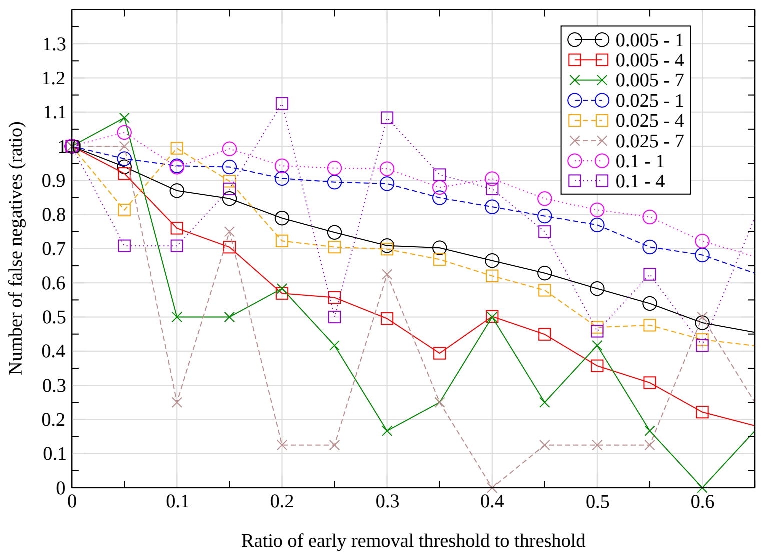

Figure 20: Effect of early removal on false negatives

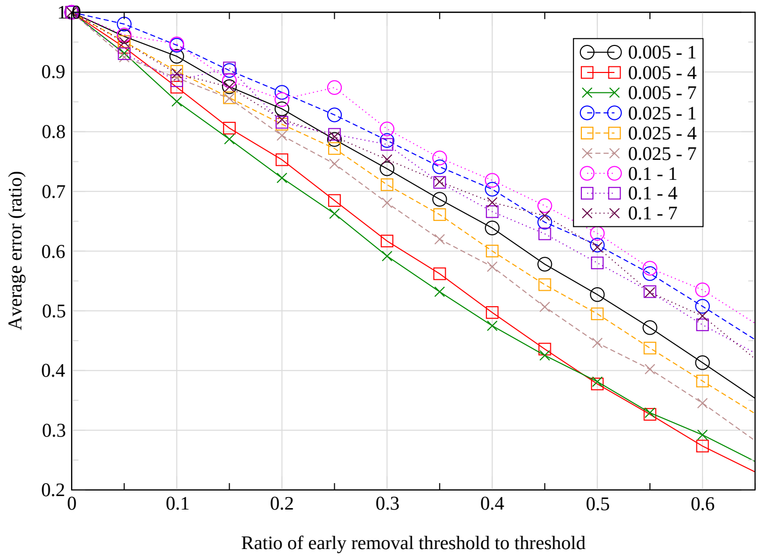

Figure 21: Effect of early removal on error

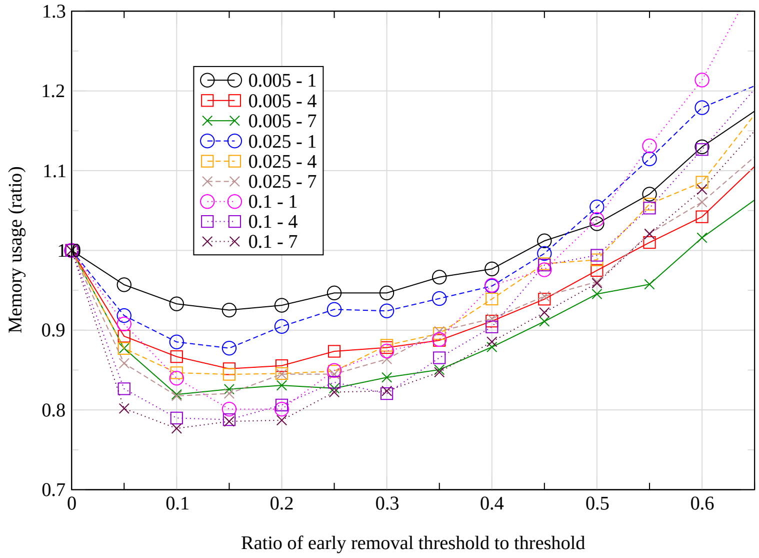

Figure 22: Effect of early removal on memory usage

Trace + flow definition	False negatives	Average error	Memory
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 11: Various measures of performance when using an early removal threshold of 15% of the threshold

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 (e.g. if the early removal threshold is one third of the threshold, we increase the oversampling by half, see Section 4.1.4 for details). The point of this experiment is to obtain the value for the early removal threshold that results in the smallest possible memory usage. Figures 20 through 22 show our results for the COS trace with 50 runs for each configuration. We can see that the probability of false negatives decreases slightly as the early removal threshold increases. This confirms that we compensated correctly for the large flows that might be removed early by increasing the oversampling. Figure 21 confirms our expectation that the average error decrease roughly linearly as the early removal threshold increases. Figure 22 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 these 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. For brevity we do not present them in full. Instead we present in Table 11 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. As in the figures, all values are reported as ratios to the values obtained without early removal.

Findings: A good value for the early removal threshold is 15% of the threshold. For this value, with oversampling is adjusted to compensate, the percentage of false negatives generally decreases slightly, the average error always decreases slightly and the memory requirements decrease typically by 20% to 30%. The decrease in memory usage is strongest when the number of flows considerably below the threshold is large. The larger the oversampling the stronger the benefits of early removal are.

G.4 Summary of findings about sample and hold

On our traces, basic sample and hold has a probability of false negatives an order of magnitude smaller than predicted in section 4.1.1. The memory requirements are also one to two orders of magnitude below what the conservative analysis predicts. Preserving entries with an early removal threshold of 15% of the threshold increases the memory requirements by rougly 20% but reduces the error in the estimates by an order of magnitude.

H Measuring multistage filters

We first compare the performance of serial and parallel multistage filters to the bound of Theorem 3. We measure the benefits of conservative update. Next we measure the effect of preserving entries and shielding. We conclude by summarizing our findings about multistage filters.

H.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

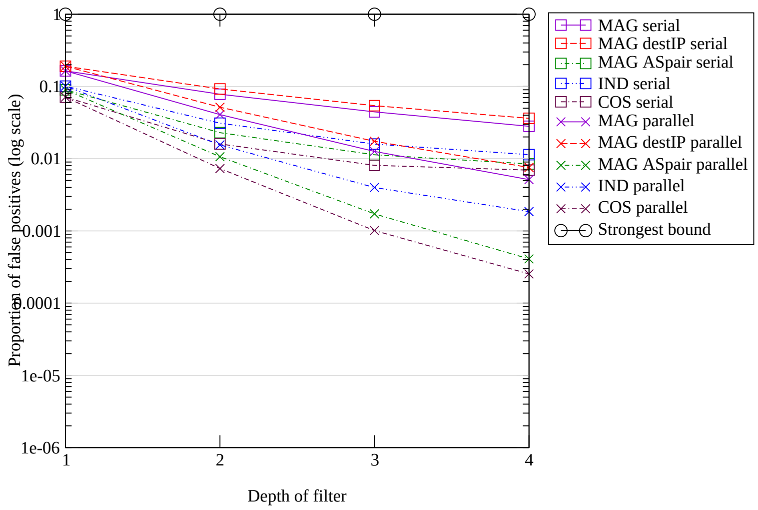

Figure 23: Actual performance for a stage strength of k=1

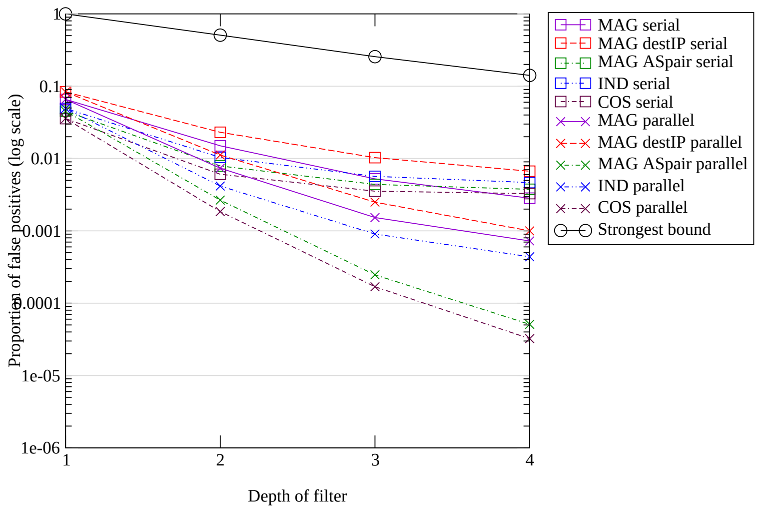

Figure 24: Actual performance for a stage strength of k=2

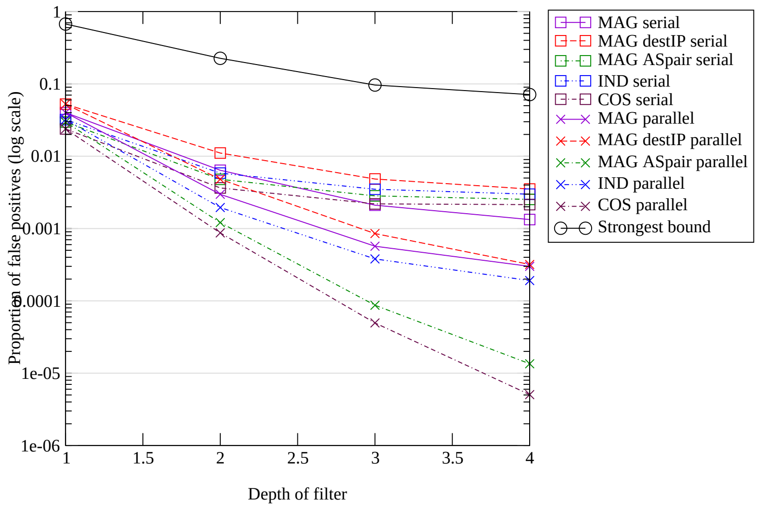

Figure 25: Actual performance for a stage strength of k=3

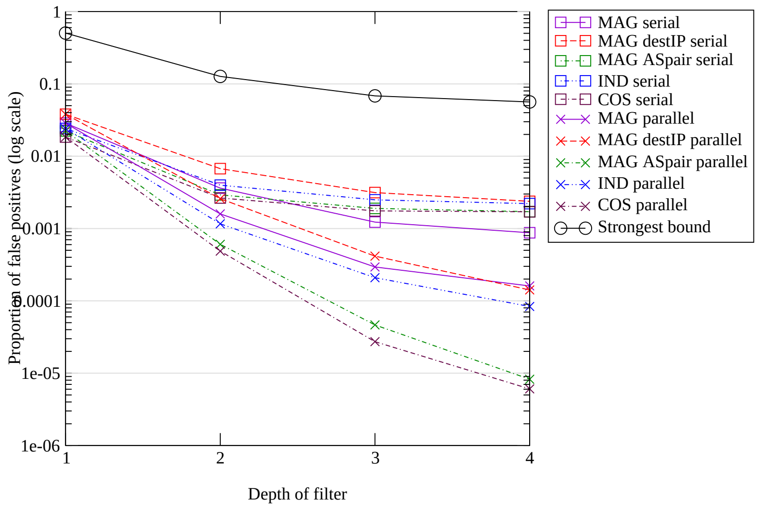

Figure 26: Actual performance for a stage strength of k=4

actually report the percentage of false positives, not their number. Another important detail is that we 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. Figures 23 to 26 present the results of our measurements for stage strengths from 1 to 4. We also represent the strongest bound we obtain from Theorem 3 for the configurations we measure. Note that the y axis is logarithmic. We can see from the results 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. This corresponds to the false positives being 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.

Findings: Multistage filters outperform Theorem 3 by up to 4 orders of magnitude (varies with the number of stages and stage strength). The percentage of false positives decreases exponentially with the number of stages. Parallel filters are much better than serial filters. The performance of the filter depends on the traffic mix.

H.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 27 and 28 the results for stage strengths of 1 and 3. The improvement introduced 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.

Findings: Conservative update reduces the number of false positives by approximately an order of magnitude (depending mostly on the number of stages).

H.3 The effect of preserving entries and shielding

Our next set of experiments evaluates the 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 average error of the estimates. The improvement in the average error does not depend much on the filter configuration. Table 12 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

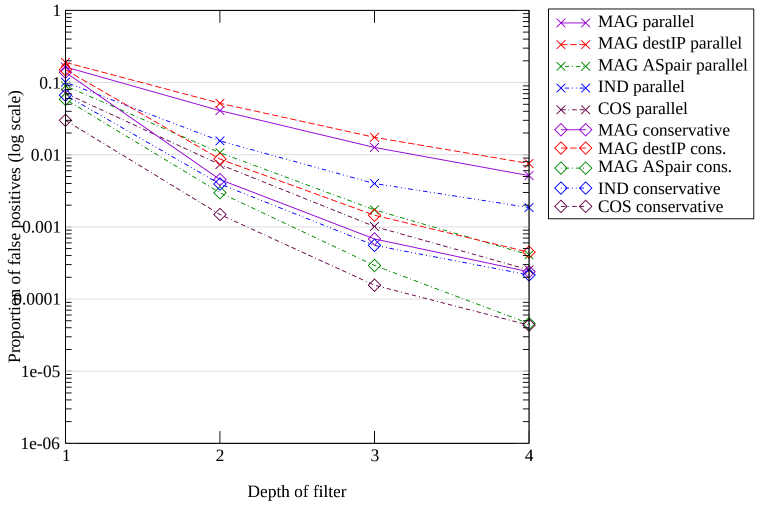

Figure 27: Conservative update for a stage strength of k=1

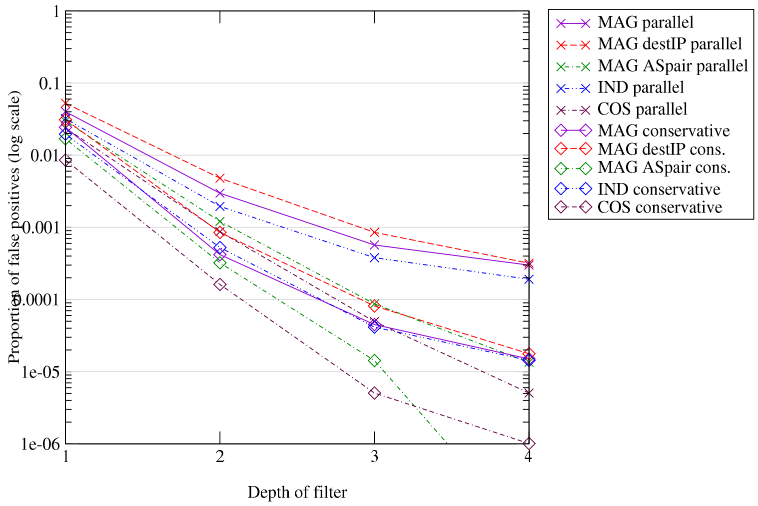

Figure 28: Conservative update for a stage strength of k=3

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 12: Average error when preserving entries compared to the average error in the base case

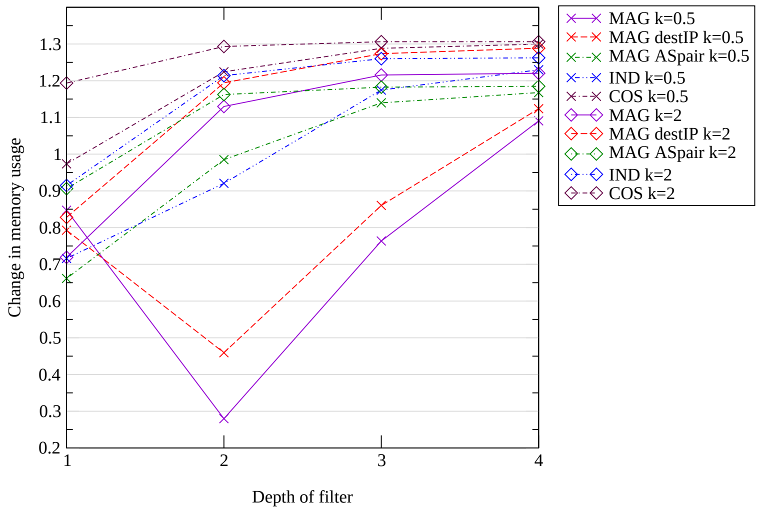

Figure 29: Change in memory usage due to preserving entries and shielding

Trace + flow ID	Sample and hold			Multistage filters		
	o=1	o=4	o=7	d=2	d=3	d=4
MAG	78.0%/92.8%	87.2%/94.4%	91.0%/95.0%	72.6%/91.3%	76.4%/92.1%	81.5%/93.0%
MAG destIP	73.6%/93.1%	88.6%/94.8%	90.2%/95.7%	65.1%/92.8%	65.7%/94.3%	85.5%/94.7%
MAG ASpair	82.3%/92.1%	87.1%/93.0%	87.8%/93.7%	63.9%/92.1%	69.5%/93.4%	70.0%/93.8%
IND	78.0%/92.5%	88.8%/94.2%	87.9%/94.4%	75.5%/91.7%	67.0%/92.4%	32.0%/92.0%
COS	83.9%/90.0%	85.7%/90.7%	86.6%/91.6%	72.1%/89.0%	66.7%/89.2%	52.1%/89.2%

Table 13: The average to maximum memory usage ratios for various configurations

of the filter (see section 4.2.3 for details) which leads to a decrease in the number of false positives. Figure 29 shows how memory usage is influenced by preserving entries. The first effect predominates for strong filters leading to an increase in memory usage while the second one predominates for weak filters leading to a decrease. The increases in memory usage are small while the improvements due to shielding can be significant. 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.

Findings: Preserving entries reduces the average error of the estimates by 70% to 85%. The effect depends on the traffic mix. Preserving entries increases the number of flow memory entries used by up to 30%. Shielding considerably strengthens weak filters. This can lead to reducing the number of flow memory entries by as much as 70%.

H.4 Summary of findings about multistage filters

Multistage filters outperform Theorem 3 by many orders of magnitude (varies with configuration and traffic mix). Parallel filters are better than serial ones and conservative update helps a lot. Shielding further increases the strength of weak filters. Preserving entries improves the accuracy of results by almost an order of magnitude (depends on traffic mix) causing an increase of up to 30% in the number of flow memory entries used.

I Calibrating the threshold adaptation algorithm

In this section we use measurements to determine the right constants to be used by the algorithm for dynamically adapting the threshold. We will determine different parameters for sample and hold and multistage filters. We first determine the safety margin and then the range of adjustment ratios.

I.1 Finding the right target usage

We use a brute force approach to finding the right measurement interval: we run the algorithms with a large number of configurations and thresholds on all traces and with all flow definitions and record the ratio between the average and maximum memory usage for each configuration. The results in table Table 13 show the minimum and average values (over all configurations). We tested thresholds between 0.005% and 1% of the link bandwidths in increments of around 40%. For sample and hold we preserved entries, used an early removal threshold of 15% and used oversampling of 1, 4 and 7. For multistage filters we used parallel filters with conservative update, preserving entries and shielding. The number of counters goes from less than the

Trace + flow ID	Perfect knowledge	Sample and hold			Multistage filters		
		o=1	o=4	o=7	d=2	d=3	d=4
MAG	0.34/1.48	1.00/1.78	1.18/1.98	1.25/2.13	0.24/7.78	0.16/10.2	0.12/12.5
MAG destIP	0.45/2.86	1.00/2.78	1.21/2.97	1.31/3.06	0.15/9.67	0.10/12.9	0.08/17.5
MAG ASpair	0.80/3.30	1.09/3.40	1.38/3.63	1.56/3.81	0.34/10.2	0.16/18.3	0.12/30.0
IND	0.95/2.27	1.23/2.97	1.38/3.64	1.35/3.76	0.35/14.0	0.17/15.9	0.17/21.4
COS	0.77/3.02	1.17/2.23	1.35/2.31	1.44/2.80	0.58/7.31	0.58/9.19	0.37/10.9

Table 14: The range of measured adjustment ratios

number of new large flows per interval for the smallest threshold up to 8 to 64 times more in increments of a factor of 2 (4 to 7 configurations) and for each number of counters we measure filters with depths of 2, 3 and 4 stages. To avoid pathological cases we do not consider the configurations where the average number of memory locations used is less than 100. We can see that for all algorithms and all traces the average ratio between the average and maximum memory usage is between 89% and 96%, but the worst case numbers are much smaller. Furthermore these numbers do not depend significantly on the number of stages or oversampling. We can also see that the minimum ratios are smaller for multistage filters than for sample and hold especially as the number of stages goes up. A conservative way to choose the target usage would be the smallest ratio seen. Since the consequence of occasional memory overflows is not that severe (especially not for sample and hold that uses early removal, so most of the entries created towards the end of the measurement interval are not reported on anyway), we use the bolder values of 90% for traffic measurement devices using sample and hold and 85% for the ones using multistage filters.

I.2 Finding the right adjustment ratios

We used the same measurements as above to get minimum and maximum values for the adjustment ratio. We it based on the ratio of the average memory usage for consecutive thresholds (approximately 40% apart). Table 14 contains our maximum and minimum values for the adjustment ratio over all thresholds and configurations. We also added the perfect knowledge algorithm (it decides which flows to add to the flow memory based on knowledge of their exact traffic) to be able to separate the effects of the peculiarities of the distributions of flows sizes from the behaviors introduced by our algorithms. We can see that sample and hold is much more robust than multistage filters (adjustment ratios closer to 1) and that it is very close (from this point of view) to the perfect knowledge algorithm. For certain settings (e.g. the MAG trace with flow ID destination IP and an oversampling of 1) it is even more robust than the perfect knowledge algorithm. We can see that the robustness of sample and hold does not depend significantly on the oversampling factor. Based on these results we use a value of 1 for *adjustdown* and 3 for *adjustup* for traffic measurement devices using sample and hold. Multistage filters have huge maximum adjustment ratios, especially when the number of stages is large. This is because when filters are overwhelmed with traffic they quickly go from strong filtering to very little filtering. Based on the results we would use the following values for *adjustdown* and *adjustup*: 0.24 and 10 for 2 stage filters; 0.16 and 16 for 3 stage filters and 0.12 and 21 for 4 stage filters. However, after a number of sample runs it turns down that these adjustment ratios are too conservative, so we use an *adjustdown* of 0.5 and an *adjustup* of 3 instead.