SIFT: A simple algorithm for tracking elephant flows, and taking advantage of power laws

Konstantinos Psounis†, Arpita Ghosh‡, Balaji Prabhakar‡, Gang Wang†
†University of Southern California, ‡Stanford University
kpsounis@usc.edu, arpitag,balaji@stanford.edu, gangwang@usc.edu

Abstract

The past ten years have seen the discovery of a number of “power laws” in networking. When applied to network trafic, these laws imply the following 80-20 rule: 80% of the work is brought by 20% of the flows; in particular, 80% of Internet packets are generated by 20% of the flows. Heavy advantage could be taken of such a statistic *if* we could identify the packets of these dominant flows with minimal overhead. This motivates us to develop SIFT, a simple randomized algorithm for indentifying the packets of large flows. SIFT is based on the inspection paradox: A low-bias coin will quite likely choose a packet from a large flow while simultaneously missing the packets of small flows.

We describe the basic algorithm and some variations. We then list some uses of SIFT that we are currently exploring, and focus on one particular use in this paper—a mechanism for allowing a router to differentially allocate link bandwidth and buffer space to large and small flows. We compare SIFT’s performance with the current practice in Internet routers via ns simulations. The advantages from using SIFT are a significant reduction in end-to-end flow delay, a sizeable reduction in the total buffer size and an increase in goodput. We comment on the implementability of SIFT and argue that it is feasible to deploy it in today’s Internet.

I. Introduction

Scheduling policies significantly affect the performance of resource allocation systems. When applied to situations where job sizes follow a heavy-tailed (or power law) distribution, the benefits can be particularly large. Consider the preponderance of heavy-tailed distributions in network traffic: Internet flow sizes [19] and web traffic [8] have both been demonstrated to be heavy-tailed. The distribution of web traffic has been taken advantage of for reducing download latencies by scheduling web sessions using the shortest remaining processing time (SRPT) algorithm. The benefit over the FIFO and the processor sharing (PS) disciplines is several orders of magnitude reduction in mean delay [12]. Given the heavy-tailed nature of Internet flow-size distribution, one expects that incorporating flow size information in router, switch and caching algorithms will lead to similar improvements in performance.

Before investigating the potential improvements in performance, we must address the following question: How can a router tell if a given packet is from a large flow? The SIFT algorithm gives a practical answer to this question, by separating (or sifting) the packets of long and short flows. SIFT randomly samples each arriving packet using a coin of small bias p. For example, with $p = 0.01$, a flow with 15 packets is going to have at least one of its packets sampled with a probability of just 0.15, while a flow with 100 packets is going to have at least one of its packets sampled with probability roughly equal to 1. Thus, most long flows will be sampled sooner or later; whereas most short flows will not be sampled. Being randomized, the SIFT algorithm will, of course, commit errors: it will likely miss some largish flows and sample some smallish flows. We later describe some simple ways of drastically reducing these errors.

Once a packet is sampled, all further packets from that flow can be processed in ways which can be beneficial. We describe one use of SIFT in this paper; Scheduling packets of long and short flows differentially. In [18], [14], another use of SIFT is discussed: caching heavy-tail distributed flows and requests at routers and web caches, respectively.

There is a large literature dedicated to servicing the packets of short and long flows differentially; see, for example, [17], [20], [11]. The paper by Gong and Matta [11] provides a detailed background and motivation for preferentially serving the packets of short flows. We summarize. Since most flows are short ("mice") and use TCP for transferring packets, dropping their packets tends to lower their effective sending rate to such a small value that they do not obtain bandwidth comparable to long ("elephant") flows. Thus, treating the packets of short flows preferentially would lead to a dramatic improvement in the *average* flow delay. The question, however, is whether such a preferential treatment of mice flows comes at the expense of significantly increased delays for elephant flows. The simulation results of [11] indicate that this is not the case. Further, Lakshmikantha et al. [13] showed via fluid-models that the benefits obtained by the short flows can be arbitrarily large while the additional delay suffered by the long flows is small and bounded. In the context of web server scheduling, [3], [5] show very similar results. And, we arrive at a similar conclusion in this paper when using SIFT.

We now consider some methods suggested in the literature for serving mice packets preferentially. There are two main approaches: classify flows as mice and elephants at the network edge and convey this to the core [11], or detect mice/elephant packets at each router locally. The first leads to different designs of edge and core routers and requires extra signalling fields in packet headers. A canonical approach in the second category advocates scheduling packets according to the least attained service (LAS) policy [17], [20]. According to this policy, the router would track the work it has *already* performed on each flow and serve the next packet from that flow which has had the smallest service till now. The logic is that when flow sizes are heavy-tailed, a small attained service is strongly indicative of a small *remaining* service due to heavy-tailed distributions being of the decreasing-hazard-rate type. Hence, mice flows will get priority. However, the LAS policy requires routers to maintain per-flow state, and to potentially maintain per-flow queues. The sheer number of flows on a high speed link [7], [10] makes this impractical.

In this paper we propose a simpler approach based on SIFT. Once SIFT identifies the packets of a large flow, they are enqueued in a low priority buffer, while the packets of short flows are preferentially served from a high priority buffer (more details later). If a packet has been put into the low priority buffer, to avoid mis-sequencing, further packets from its flow will also be put into the low priority queue.[1]

We use ns to understand how well SIFT performs in reducing delays. We find that, depending on the load, SIFT reduces the average delay of short flows between one and two orders of magnitude, and the overall average delay between two and ten times. The delays of the very large flows are worsened by a factor less than 2. To avoid starvation of the long flows, we change the basic scheme and guarantee the low priority buffer a small bandwidth. We find that this does not affect the performance significantly (although, as we shall see, it introduces a resequencing problem).

An interesting finding of our study is that SIFT can *reduce total buffer requirements* by a factor of two without any performance degradation. That is, both the goodput and flow delays can be competitive or slightly improved using the two-buffer SIFT system as compared with a single FIFO of twice the total size. In the paper we explore reasons for this gain.

The organization of the paper is as follows: In Section II we describe SIFT in detail. Section III presents the simulations results with TCP flows using ns-2 [15] and Section IV deals with implementation and deployment issues. Finally, Section V describes further work.

[1]This is reminiscent of the "sample and hold" strategy advanced in [6] for identifying and counting the packets of high bandwidth flows. Indeed, the SIFT algorithm can be considered as a "dual" of the sample and hold algorithm in that it discriminates flows on the basis of their size, while the sample and hold algorithm does this on the basis of rate. We see later how both can be used in conjunction.

II. The SIFT Algorithm

For each arriving packet we toss a coin with bias p for heads.[2] Coin tosses are independent of all else. We say that a flow is "sampled" if one of its packets gets a heads on it. Thus, a flow of size X is sampled with probability $P_S(X) = 1 - (1-p)^X$. Suppose that $p = 0.01$. Then, this scheme is a randomized version of a deterministic scheme which does not sample *any flow* with fewer than 100 packets and samples *all flows* with more than 100 packets. The deterministic scheme would require counters for each flow (large and small), making it expensive at core routers. By contrast, SIFT mostly tracks the large flows. However, being randomized, SIFT commits errors. That is, a flow with size less than 100 is sampled with positive probability whereas a flow with size larger than 100 is not sampled with positive probability.

To compensate for the errors, we consider the following refinement, called SIFT-2Toss: Say that a flow is sampled if *two* of its packets get heads on them when we use a coin of bias $2p$. Thus, the probability that a flow of size X is sampled equals $1-(1-2p)^X - 2pX(1-2p)^{X-1}$. It is easy to see that this reduces the errors significantly. Due to a lack of space we omit this calculation here; it can be found in [16].

Let us determine precisely what the error is for SIFT when $p = 0.01$ and we are sampling flows with a bounded Pareto distribution on the number of packets. Recall that under the bounded Pareto distribution, the flow size variable X is distributed as:

$$P(X = x) = cx^{-\alpha-1}, \quad \text{for } m \leq x \leq M$$

where c is a normalization constant and $\alpha \in (1, 2)$. Taking $\alpha = 1.1$, $m = 1$ and $M = 10^6$, the probability of error given by

$$P_e = \sum_{x<\frac{1}{p}} P_S(X)P(X = x) + \sum_{x\geq\frac{1}{p}} (1 - P_S(X))P(X = x)$$

evaluates to 2.6%. That is, the fraction of flows that are misclassified is only 2.6%. This small error is not a real surprise and is a consequence of the heavy-tailed distribution: the short flows are really short and the long ones are really long. Therefore, we see that the basic SIFT algorithm is already good enough, we don't need to consider SIFT-2Toss.[3]

Having described the SIFT sampling algorithm, we use it for serving the packets of short flows preferentially as described in the Introduction. That is, we maintain two queues, separately queue the packets of sampled and unsampled flows, and serve the queues with different priority. If the original queue has size B, we do not let the combined size of the high and low priority queues to exceed B. Figure 1 presents SIFT schematically. In the next section we evaluate the performance of SIFT using extensive simulations. For a queueing theoretic analysis of SIFT, we refer the interested reader to [16].

Remark: SIFT-based buffering of packets leads to a flow scheduling mechanism; hence, clearly, it has implications for congestion control. To put this more sharply, consider the deterministic version of SIFT applied to the router buffering problem. Such a scheme would use a size threshold T, and forward up to the first $T-1$ packets of *all* flows to the high priority queue, and the rest of the packets to the low priority queue.[4] Then, depending on the value of T, this scheme would treat flows during their slow-start phase preferentially compared to when they enter the congestion-avoidance phase. (Of course, mice flows predominantly do not enter congestion-avoidance.) The connection with congestion control is now clear.

[2]Note that we toss coins on packets, not bytes. It is easy to modify the scheme, but difficult to describe it, when taking into account the variable nature of packet sizes. We omit this variation here.

[3]We mention SIFT-2Toss because it allows the possibility of changing the bias on each toss. For example, instead of tossing a coin with bias $2p$ twice, we could toss coins with biases p_1 and p_2 so that $\frac{1}{p_1} + \frac{1}{p_2} = \frac{1}{p}$.

[4]We note that the deterministic scheme has been previously considered; see, for example, the Size-aware TCP Scheduler (SaTS) web-page at `http://www.cs.bu.edu/fac/matta/` and the references therein.

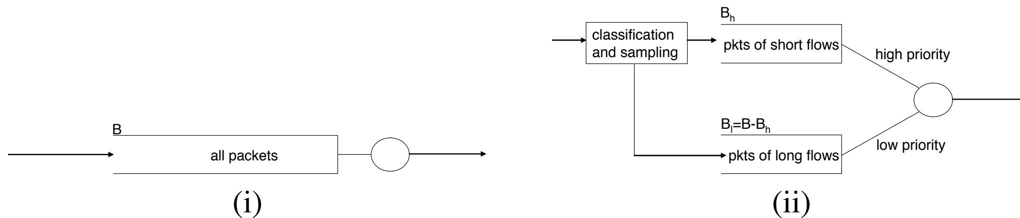

Fig. 1. (i) The original FIFO queue, (ii) SIFT-based queues.

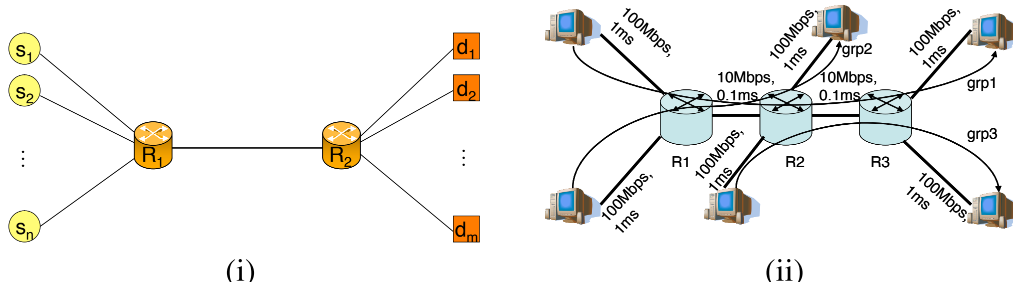

Fig. 2. (i) Simple network topology, and (ii) network topology with multiple congested links.

III. Simulations

This section evaluates the performance of SIFT in an Internet-like environment using ns-2 simulations [15]. We shall consider various senarios, as described below.

A. *Single congested link*

Consider a single congested link topology as shown in Figure 2(i). There are n source nodes, m destination nodes, and two internal nodes, R_1 and R_2. The link capacity and the propagation delay between the source/destination nodes and internal nodes will vary from experiment to experiment. The capacity of the link between nodes R_1 and R_2 is 10Mbits/s and the propagation delay of this link is set to 0.1ms. SIFT is deployed on this link.

The two queues of SIFT and the strict priority mechanism between them is implemented using the CBQ functionality of ns-2. We also run simulations where the low priority queue is guaranteed a proportion of the link capacity. We call this scheme SIFT$_{GBW}$. Each of the two queues use either DropTail or RED. Their size is set to 100 packets. When SIFT is not used, the corresponding single queue has size equal to 200 packets. (Later we study the case where the two queues have different sizes that might add up to something less than 200.) Throughout the experiments we use a sampling probability equal to 0.01.

The traffic is generated using the built-in sessions in ns-2. A total of 300,000 web sessions are generated at random times in each experiment, each session consisting of a single object. This object is called a flow in the discussion below. All flows are transfered using TCP. Each flow consists of a Pareto-distributed number of packets with flow sizes varying between 1 and 10^8, and shape parameter equal to 1.1. (This is motivated by the well-known result that the size distribution of Internet flows is heavy-tailed, see, for example, [2], [19].) The packet size is set to 1000 bytes.

By varying the rate at which sources generate flows, we study the performance of SIFT at various levels of congestion. In the rest of the section we characterize the congestion level of each experiment by the traffic intensity in the bottleneck link, defined as $\rho = \lambda E(S)/C$, where λ is the aggregate average arrival rate, $E(S)$ is the average flow size, and C is the

link capacity. The percentage of drops in the experiments varies from 0 to 9% depending on the load.

1) Basic setup: We start with the simplest setup, where we have just one source destination pair ($n = m = 1$). The link capacities between the source/destination and the internal nodes are 100Mbits/s and the propagation delay of these links equals 1ms.[5]

We first present results when RED is used. In this experiment, we find that approximately 95% of all flows are "short", i.e., they are never sampled, and 5% of all flows are "long". (This fraction is, of course, a function of the sampling probability as well as the job size distribution; we have not attempted to optimize the sampling probability for the best average delays.) Figure 3 compares the short-flow and long-flow average delay with and without SIFT for $\rho = 0.6$, 0.7, 0.9, 1.2, and 1.5.[6] It is evident from the plot that with SIFT, the short flows can have a delay close to two orders of magnitude less than without SIFT. This gain does not significantly increase the delay of the rest of the flows: the average delay of long flows is at worst doubled under SIFT. Finally, note that the overall average delay of flows exhibits a 2x to 4x improvement, depending on the load (see Figure 5). The results are nearly identical when DropTail is used instead of RED.

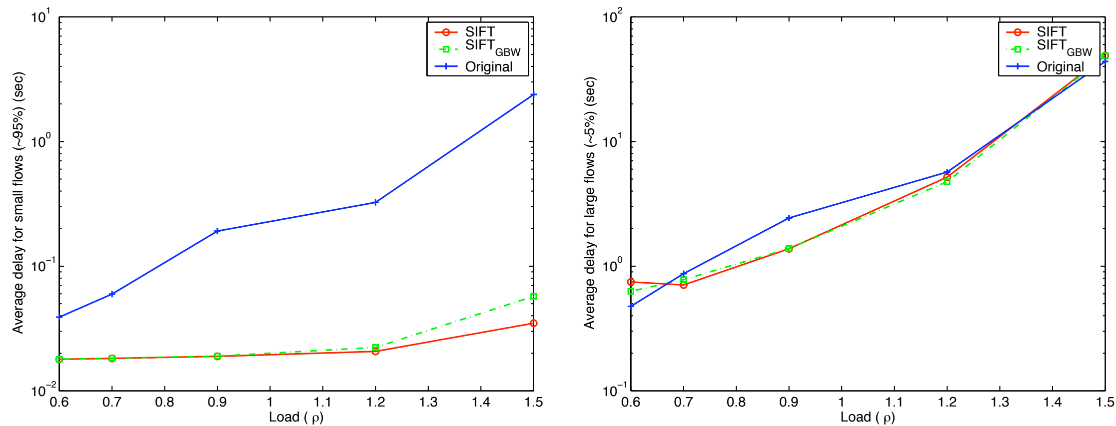

Fig. 3. Small-flow and large-flow delay under SIFT.

2) Guaranteeing bandwidth for long flows: We also run simulations where a fraction of the total link bandwidth (10% in this particular experiment) is dedicated to the low priority queues containing the long flows. This ensures that long flows are not starved. Figure 3 shows that the performance of SIFT$_{GBW}$ is very similar to that of SIFT which uses strict priority. As expected, the average delays for long flows are smaller when bandwidth is guaranteed to the low priority queue, and the average delays for short flows are larger. It is evident that the difference between the delays is small enough to be negligible. So, at least on average, we need not fear starvation for long flows with SIFT. Alternatively, if it is a concern, we could implement SIFT$_{GBW}$.

Note that under this scenario packet reordering is possible. This occurs when the last packet to join the high priority queue from a long flow, leaves the router after subsequent packets of the flow. However, this is very unlikely to occur since subsequent packets join

[5] In order to better observe the effect of SIFT on queueing delay, we choose a small propagation delay here. This makes queueing delay the dominant component of total delay . Later in this section we also present results corresponding to larger propagation delays.

[6] Having ρ exceed 1 is deliberate. We want to observe the system under extremely high loads so as to see the effectiveness of SIFT in helping small flows. In this case the average delay values are taken to correspond to those flows which complete transfer. We note here that the number of flows not completing transfer (i.e. timed-out) are comparable under the SIFT and FIFO schemes.

the low priority queue, and this queue is not only larger on average, but also is serviced by a very small fraction of the total link capacity.

3) Other scenarios: We investigate how SIFT performs in a variety of situations. Since the results are very similar to those of Figure 3, and due to limitations of space, we do not present the corresponding plots. The plots can be found in [16].

First, we investigate how SIFT behaves towards *fast and slow flows*. Suppose that the following two kinds of flows share a link: *Turtles*, with low bit rates, and *Cheetahs*, with high bit rates. Note that it is possible to have a long turtle and a short cheetah; the former is a large-sized flow from which packets arrive at a low rate, while the latter is a small-sized high rate flow. A worrisome thought in this case is that SIFT would further slow down the already slow and long turtles, since the sampling mechanism here is packet based, and hence responds only to flow sizes without taking flow rates into account. (A *rate-based* (rather than a size-based) sampling scheme, such as the one suggested in [6], could be used if differentiation based on rate is also desirable.) We use a simulation scenario with two source-destination pairs, both passing through the bottleneck link. One link, on which turtle flows arrive, has a speed of 5Mbits/s and a propagation delay of 1ms, and the other, on which cheetah flows arrive, runs at 495Mbits/s and has propagation delay equal to 0.1ms. The speed and the propagation delay of the bottleneck link, as before, is 10Mbits/s and 0.1ms respectively. The simulations show that the overall average delay for turtles with SIFT is less than that without SIFT. Also, the average delay for long turtles is similar to the previous cases; that is, long turtles are not hurt by SIFT.

Second, we study how SIFT behaves towards *different propagation delays*. The worry here is that large flows with long RTTs might be excessively penalized. We run simulations with three source destination pairs ($n = m = 3$). For the first source-destination pair, we use a propagation delay of 0.1ms on the links from source/destination to the internal nodes; 1 ms on each link for the second, and 5 ms for the third pair. The speeds of these links are all 100Mbits/s. The simulations indicate that while the delays are higher for the flows with large RTTs than for the ones with small RTTs, SIFT consistently provides gains for all three classes of flows.

Third, we consider the case where we have *a large number of slow links feeding the bottleneck link*. This is meant to capture the case, for example, when a large number of users connect to the Internet through slow modem links. We simulate this scenario using 10 source destination pairs, with each input and output link having a speed of 3Mbits/s. The bottleneck link has a speed of 10Mbits/s, as usual. In this case also, the use of SIFT leads to significant improvements in performance for most flows, at the expense of only a small increase in the delay of long flows.

B. Multiple congested links

So far simulations were conducted in a topology with a single bottleneck link. However, it is well documented that Internet flows often go through multiple bottlenecks. This raises the following natural question: Should SIFT be employed at one of the congested links along the path of a flow or at all of them? And if it is deployed at more than one link, would long flows be significantly delayed? To answer these questions, we perform simulations in the setup shown in Figure 2(ii) that consists of two bottleneck links.

In this setup there are three groups of flows. The first goes through the first congested link, the second goes through the second congested link, and the third goes through both links. We run simulations where SIFT is employed in the first link only (SIFT_1) or in both links (SIFT_2). Figure 4 shows the short-flow and long-flow delay for the third group which is the group of interest. As expected, SIFT_2 yields lower short-flow delays than SIFT_1. Interestingly enough,

this occurs without excessively penalizing the long flows; the long-flow delay is quite similar under all schemes. For completeness, in Figure 5 we also show the overall average delay for both the basic single congested link scenario and the multiple congested link scenario. As expected from the short-flow and long-flow delays already presented, SIFT improves the overall average delay sizeably in both scenarios.

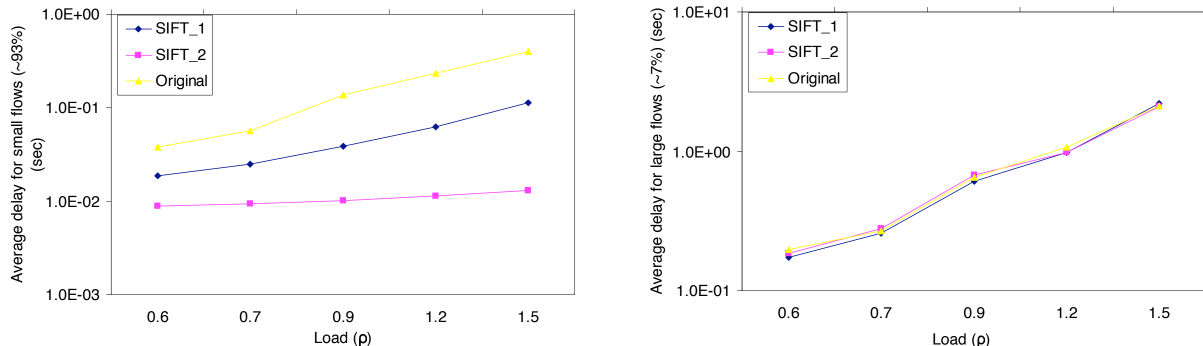

Fig. 4. Small-flow and large-flow average delay under SIFT_1 and SIFT_2.

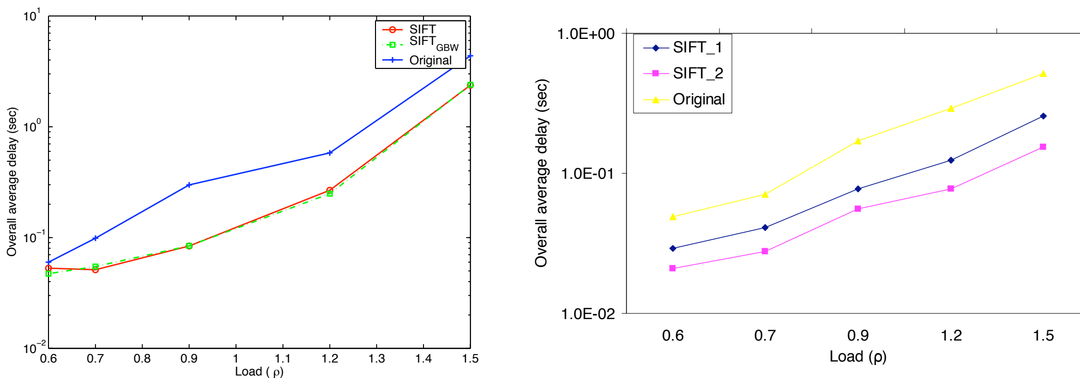

Fig. 5. Overall average delay in basic single congested link scenario and multiple congested link scenario.

A comment on what constitutes a long flow in the multiple bottleneck case is in order. Clearly, the flows sampled under SIFT_1 are not the same as the flows sampled under SIFT_2. For this reason, we define a flow as "long" if it exceeds a size threshold, which in our case is equal to the inverse of the sampling probability; that is, 100. As we explain in [16] this is not an arbitrary choice. It equals the expected number of packets until a flow is sampled.

C. Buffer savings

Unequal partitioning of the total buffer space between the two queues might yield better performance than setting $B_h = B_l = B/2$. For example, since the high priority queue has strict priority, one expects its buffer space requirement to be relatively low, and there is no need to allocate $B/2$ buffer space to it.

Figure 6 plots the instantaneous queue sizes when DropTail is used and $\rho = 0.9$. It is evident that if the high priority queue had a size of 50 instead of 100, the results would have been identical. This is true not because the system is never congested; the original queue does get full at some point in time. For larger ρ, this is even more pronounced. For example, for $\rho = 1.2$, the original and the low priority queues are nearly always full, while the high priority queue has never more than 60 packets. Interestingly, this is a general trend in all

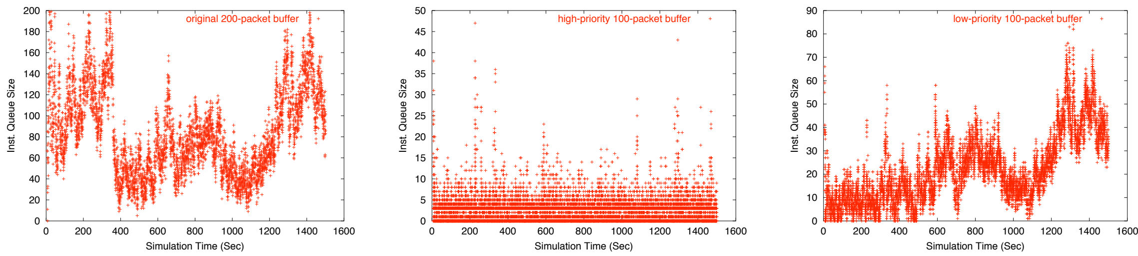

Fig. 6. Instantaneous sizes of the original 200-packet buffer, the high-priority 100-packet buffer, and the low-priority 100-packet buffer.

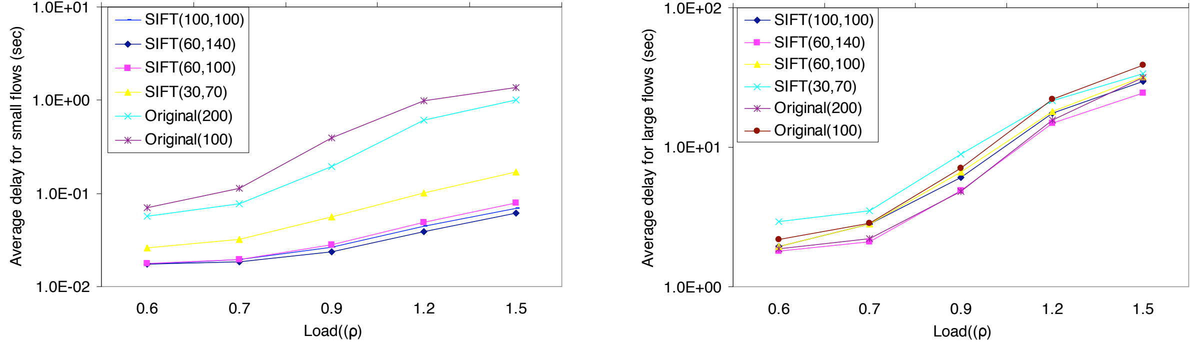

Fig. 7. Small-flow and large-flow average delay under SIFT for various buffer allocations.

experiments: the original 200-packet and the low priority 100-packet queue face similar levels of congestion, while the high priority 100-packet queue faces significantly lower congestion or no congestion at all. (Due to limitations of space we don't present more plots with queue dynamics.)

Based on the above observations, it is evident that in addition to significantly reducing average flow delays, SIFT also leads to sizable buffer savings at the packet level. To investigate this further, we run simulations in the simple topology depicted in Figure 2(i) using versions of SIFT for which $B_h + B_l < B = 200$. (We denote such a version by SIFT(B_h, B_l).) As evident from Figure 7, SIFT's performance remains virtually unchanged for a total buffer space equal to 160 and it is slightly worse for a total buffer space as low as 100.

One way to explain SIFT's lower buffer requirement is the following. Intuitively, short flows enter a fast lane and their service is completed very fast, and it is only the packets of long flows that have to be buffered. Now buffer sizes should be kept large enough to ensure high utilization: This is the well-known bandwidth-delay principle, according to which the buffer size should equal $C \times RTT$. [7] However, this rule *only* applies to the long flows. SIFT separates the packets of short flows from those of the long flows. The arrival rate of the latter is a fraction of the total capacity, call it $C' < C$. Hence, a buffer size equal to $C' \times RTT$ is enough. In our basic scenario, the proportion of traffic arriving to the low-priority buffer equals 56% (it is important to keep in mind that the initial packets of the long flows do not arrive to this buffer, else the percentage would be larger than 56).

The only metric considered so far is the average delay. We now present results for two more metrics. First, we show histograms of delays to better understand how SIFT alters the delays of flows. Second, we show the total number of packets dropped from the congested link of Figure 2(i). Figure 8(i) shows the delay histogram under various schemes for $\rho = 1.2$.

[7]Recently, this rule-of-thumb has been corrected to $\frac{C \times RTT}{\sqrt{n}}$ where n is the number of long flows going through the router [1].

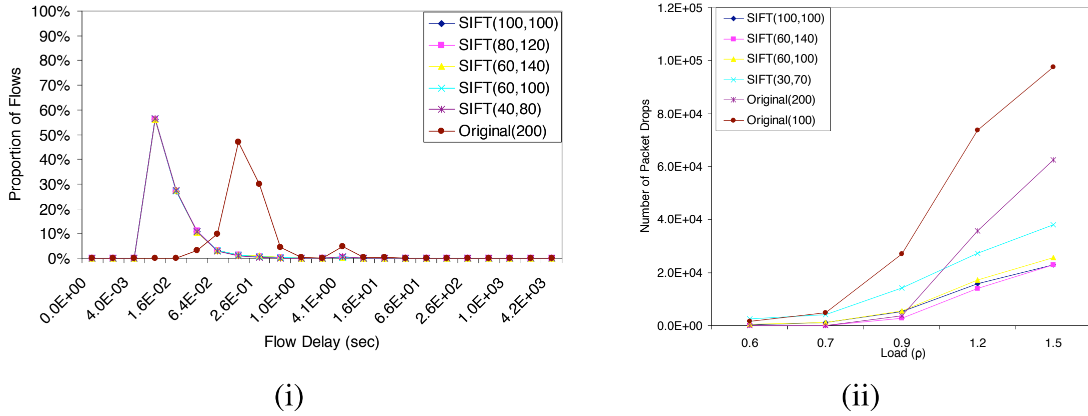

Fig. 8. (i) Histogram of flow delays for $\rho = 1.2$, and (iii) total packet drops in the congested link.

(The leftmost point in the plot corresponds to flows with total delay between 0 and 2ms. The second to flows with delay between 2 and 4ms, the third between 4 and 8ms, the fourth between 8 and 16ms and so on.) As it is evident from the plot, the histogram under SIFT remains virtually unchanged for the majority of flows, as one reduces the total available buffer from 200 down to 120. Notice that the large difference between the peaks of the histograms under SIFT and the original scheme is due to the large queueing delay that all flows face under the latter scheme. Figure 8(iii) shows the total number of drops as a function of the traffic intensity. For traffic intensities larger than 0.9 where the drop probability is sizable (larger than 1%), SIFT incurs significantly lower drops than the original scheme.

IV. Implementation and Deployment

In this section we asess the implementation requirements of SIFT and comment on its deployability. Compared to a traditional linecard, a SIFT-enabled linecard requires the following additions: (i) maintaining two queues instead of one in the physical buffer, (ii) implementing strict priority between the two queues, (iii) sampling packets at the time of their arrival, (iv) maintaining the flow id of the sampled packets, (v) identifying the flow to which a packet belongs to, and (v) evaluating whether a flow is sampled or not.

The first two requirements are clearly very simple. Maintaining two queues only requires a second linked list, and strict priority only requires checking whether the head of the list corresponding to the high priority queue is null or not. Also, note that circular buffers may be used to make it possible to “lend” unused buffer space to the other queue. The last two requirements are also very easy to fulfill. There are widely available mechanisms today in commercial routers which identify the flow to which a packet belongs. (Notice that in accordance with usual practice [7], [9], [10], packets are said to belong to the same flow if they have the same source and destination IP address, and source and destination port number.) And, standard hashing schemes can be used to evaluate if a flow is sampled or not at a very low cost.

Sampling packets is very inexpensive. A pseudo-random number generator can be used to decide whether to sample or not, or one may choose to “sample” every, say, 100th packet. The later is a common practice to avoid random sampling; for example, it is used in Cisco’s Netflow monitoring tool [4].

Finally, keeping the id of every sampled flow might be problematic if the number of sampled flows is too large. However, the heavy-tailed nature of flow sizes is directly helpful here: the number of large flows (and hence the number of sampled flows) is small. Additionally, aging the entries removes them from the table and helps keep the number of active sampled

flows small. In all our simulations this number was not more than 1,500 and usually much smaller, even though the number of active flows was about 10 times as large.

V. Further work

We have described the SIFT algorithm and its use in differentially serving the packets of short flows. There are many interesting questions to pursue further; notably, that of understanding the interaction of SIFT-based buffering with end-to-end congestion control, and the buffer size reduction phenomenon.

It is our view that the SIFT sampling mechanism is of interest for taking advantage of the 80-20 rule induced by power law distributions. As such, it can help network systems improve their performance significantly and might apply in several other situations. It is, therefore, worth investigating these situations in some detail. At a high level, network systems such as routers, switches and web caches treat packets and requests individually, *regardless* of whether a particular packet (or request) belongs to a large or a small flow. In the presence of an 80-20 rule, taking advantage of flow information could be very beneficial to system performance.

References

[1] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *Proceedings of the ACM SIGCOMM Conference*, 2004.

[2] Sprint ATL. Sprint network traffic flow traces. `http://www.sprintlabs.com/Department/IP-Interworking/Monitor/`, accessed 2002.

[3] Nikhil Bansal and Mor Harchol-Balter. Analysis of SRPT scheduling: investigating unfairness. In *ACM SIGMETRICS/Performance*, pages 279–290, 2001.

[4] Cisco. NetFlow services and applications. White paper, 2000. `http://cisco.com/warp/public/cc/pd/iosw/ioft/neflct/tech/napps_wp.htm`, accessed January 2002.

[5] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *Proceedings of USITS*, 1999.

[6] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of ACM SIGCOMM*, 2002.

[7] W. Fang and L. Peterson. Inter-as traffic patterns and their implications. In *Proceedings of the 4th Global Internet Symposium*, December 1999.

[8] A. Feldmann. Characteristics of TCP connection arrivals. In *Self-Similar Network Traffic and Performance Evaluation*, K.Park and W.Willinger, editors, Wiley, 2000.

[9] C. Fraleigh, C. Diot, B. Lyles, S. Moon, P. Owezarski, D. Papagiannaki, and F. Tobagi. Design and deployment of a passive monitoring infrastructure. In *Proceedings of the Workshop on Passive and Active Measurements, PAM*, April 2001.

[10] C. Fraleigh, S. Moon, C. Diot, B. Lyles, and F. Tobagi. Packet-level traffic measurements from a tier-1 IP backbone. Technical Report TR01-ATL-110101, Sprint ATL Technical Report, November 2001.

[11] L. Guo and I. Matta. The war between mice and elephants. In *Proceedings of the ICNP*, 2001.

[12] M. Harchol-Balter, N. Bansal, B. Schroeder, and M. Agrawal. Implementation of srpt scheduling in web servers. Technical Report CMU-CS-00-170, Carnegie Mellon Universit, 2000.

[13] A. Lakshmikantha, R. Srikant, and C. L. Beck. Processor sharing versus priority schemes for TCP flows in Internet routers. In *Proceedings of Conference on Decision and Control*, 2005.

[14] Y. Lu, D. Mosk-Aoyama, and B. Prabhakar. An analytic study of the seive cache. In Preparation, 2004.

[15] Network simulator. `http://www.isi.edu/nsnam/ns`, accessed June 2003.

[16] K. Psounis, A. Ghosh, and B. Prabhakar. Sift: A low-complexity scheduler for reducing flow delays in the internet. Technical Report CENG-2004-01, University of Southern California, 2004.

[17] I. A. Rai, G. Urvoy-Keller, and E. W. Biersack. Analysis of LAS scheduling for job size distributions with high variance. In *Proceedings of the ACM SIGMETRICS Conference*, 2003.

[18] M. Wang, X. Zou, F. Bonomi, and B. Prabhakar. The seive: A selective cache for heavy-tailed packet traces. In Preparation, 2004.

[19] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through high-variability: Statistical analysis of ethernet lan traffic at the source level. *IEEE/ACM Transactions on Networking*, 5(1):71–86, 1997.

[20] U. Madhow Z. Shao. Scheduling heavy-tailed traffic over the wireless internet. In *Proc. IEEE Vehicular Technology Conference*, September 2002.