FineComb: Measuring Microscopic Latency and Loss in the Presence of Reordering

Myungjin Lee, Sharon Goldberg, Ramana Rao Kompella, and George Varghese

***Abstract*—Modern stock trading and cluster applications require microsecond latencies and almost no losses in data centers. This paper introduces an algorithm called *FineComb* that can obtain fine-grain end-to-end loss and latency measurements between edge routers in these networks. Such a mechanism can allow managers to distinguish between latencies and loss singularities caused by servers and those caused by the network. Compared to prior work, such as Lossy Difference Aggregator (LDA), that focused on switch-level latency measurements, the requirement of end-to-end latency measurements introduces the *challenge of reordering* that occurs commonly in IP networks due to churn. The problem is even more acute in switches across data center networks that employ multipath routing algorithms to exploit the inherent path diversity. Without proper care, a loss estimation algorithm can confound loss and reordering; further, any attempt to aggregate delay estimates in the presence of reordering results in severe errors. FineComb deals with these problems using order-agnostic packet digests and a simple new idea we call stash recovery. Our evaluation demonstrates that FineComb is orders of magnitude more accurate than LDA in loss and delay estimates in the presence of reordering.**

***Index Terms*—Passive measurement, latency, packet loss, reordering, algorithms.**

I. Introduction

Recent trends in data centers have led to requirements for *microsecond* latencies. Fundamentally, this is because programs respond to network messages, not humans. For example, an automated trading program can buy millions of shares cheaply with faster access to a low stock price; similarly, a cluster application can execute 1000s more instructions if latencies are trimmed by 100 μsecs. Further, all these applications are deployed in data centers that span a small geographical area and where links and switches are carefully chosen to have minimal latencies (*e.g.*, [1]). It is unlikely that this trend toward low latency networks is going to stop any time soon; indeed, analysts are already discussing applications that would require even more stringent latency guarantees in the order of nanoseconds [2].

Portions of this manuscript appeared in ACM SIGMETRICS 2011. We thank Kirill Levchenko, Michael Mitzenmacher, and Zvika Brakerski for comments on previous versions of this manuscript. This work was supported in part by NSF Award CNS 0831647, 0964395, 1054788, and a grant from Cisco Systems.

M. Lee is with the School of Informatics, University of Edinburgh, Edinburgh EH8 9AB, UK (e-mail: myungjin.lee@ed.ac.uk).

S. Goldberg is with the Department of Computer Science, Boston University, Boston, MA 02215 (e-mail: goldbe@cs.bu.edu).

R. R. Kompella is with the Department of Computer Science, Purdue University, West Lafayette, IN 47907 (e-mail: kompella@cs.purdue.edu).

G. Varghese is with the Department of Computer Science and Engineering, UCSD, La Jolla, CA 92040 (e-mail: varghese@cs.ucsd.edu).

Despite the most careful selection of network components, there is no easy way for network operators to guarantee that congestion in switches never causes latencies to increase beyond acceptable thresholds. For example, in the well-known "in-cast" problem [3], the effects of barrier-synchronized workloads overflow switch buffers and lead to packet loss and high latency. While solutions and workarounds may often exist for specific problems, operators need to measure latencies on a continuous basis to detect and fix such problems, for instance, by re-routing the offending application.

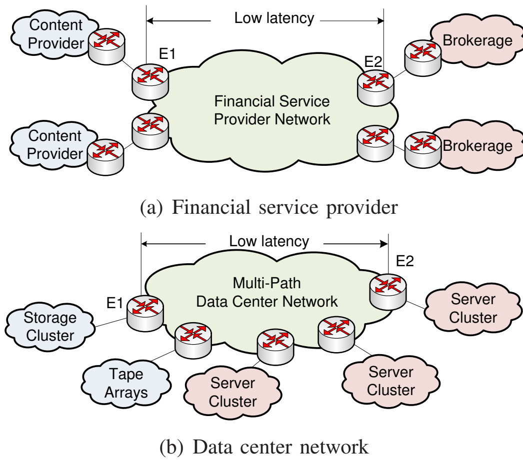
At a minimum, network operators typically need two types of measurements. First, they need *end-to-end* [1] *measurements* in the network to check if end-to-end latencies and losses are within satisfactory limits for a given customer that are often specified in the form of service-level agreements (SLAs). Second, if a customer experiences bad performance, it is important to quickly diagnose the root cause of the problem by obtaining *switch-level measurements* to localize the offending switch. While solutions such as LDA [4] have been recently proposed for measuring switch-level delays, detecting end-to-end latency spikes with LDA often scales poorly since each switch needs to be equipped with it. We therefore focus on a new end-to-end latency measurement solution.

In this paper, we consider passively measuring end-to-end delays of actual packets that travel between two endpoints. This approach results in two immediate benefits. First, it does not interfere with regular traffic. Second, SLA violations apply to actual packets; so, measuring actual packet latencies will reflect the SLA violations better than using artificial probes.

Depending on the particular scenario, the two endpoints between which latency and loss measurements are required vary. For example, in a market data network [5] (Figure 1(a)), data feeds from content providers (*e.g.*, stock exchanges) are often provided to individual brokerages using financial service providers (FSPs). Here, the FSPs may want to provide a latency SLA of a few μseconds through their network from the content provider to the brokerage; hence measurements between these edges are crucial. In a typical data center network running low-latency applications, clusters of servers are interconnected with storage servers, tape arrays and other such infrastructure [6] (Figure 1(b)). In such cases, one could easily imagine stringent latency requirements between server and storage cluster, or across two different server rack switches, within a multi-rooted tree topology (*e.g.*, a fat-tree [7]).

In our end-to-end setting, we need to allow for the presence

[1] In our context, it means all sub-paths between two measurement endpoints, not an end-to-end path for a flow.



(a) Financial service provider

(b) Data center network

Fig. 1. Low end-to-end latency applications.

of *packet reordering* across all sub-paths between the two measurement endpoints. While switch vendors typically ensure that there is no reordering across flows between two interfaces (otherwise, TCP may not work well), no such guarantee is provided by an IP network across routers that are not directly connected. In fact, many commercial data centers rely on exploiting the path diversity inherently present within data centers using ECMP (equal cost multipath) where flows are split across multiple paths. Of course, while ECMP still ensures packets within a flow are not reordered, reordering commonly occurs across flows. In addition, churn in the network (*e.g.*, link failures) can cause temporary routing loops that may introduce reordering.

Furthermore, while our immediate motivation is end-to-end reordering that can happen in IP networks, we believe it is very likely that future switches will allow reordering within switches for improved load-balancing using techniques like packet spraying. Standard TCP implementations then interact poorly with reordering in that they reduce window sizes unduly by conflating loss and misordering. However, a number of researchers have been looking at creating reordering-tolerant TCP such as Multipath-TCP [8], at least for use in data centers. While these ideas appear radical, packet spraying with reordering-tolerant TCP at the edges can greatly improve the utilization and costs of future data center networks. If these ideas gain currency, as we believe they will, making scalable latency measurement resilient to reordering will be essential not just end-to-end, but also within switches.

The state-of-the-art solution LDA [4], which assumes FIFO packet ordering, will not work well in these environments, as it can confuse reordered packets with lost ones. To address this problem, this paper describes an efficient data structure called FineComb that is robust to reordering, and can be easily implemented at the edges to spot microscopic delay variations (in the order of μseconds) and losses (10s per million) with small amount of state and processing costs. We evaluate FineComb extensively both analytically, and via simulation on various delay models and real router traces; our experiments indicate that compared to LDA, FineComb can achieve 10x and 200x lower relative errors for latency and loss estimates, respectively, even under small amounts of reordering.

II. Preliminaries

We describe the basic measurement goals, constraints and assumptions in our problem setting, and explain a set of existing solutions that do not work well for our problem.

A. Measurement Goals

Our goal is to measure the aggregate performance across all sub-paths between two edge routers, say $E1$ and $E2$ in Figure 1. We divide time into intervals (a few seconds) for which we are keen to obtain performance metrics. We consider three basic measures across all packets: average latency, variance, and loss rate.

For most of this paper, we assume hardware implementations to keep up with high line rates; however, we briefly discuss software implementations. We require that our data structure scales well in terms of control bandwidth, processing time, and storage. While storage may possibly be increased in a software implementation, processing time and control bandwidth need to be kept a minimum. Further, as we mentioned before, the solution should be robust to packet reordering that may occur in these environments.

B. Assumptions

We make three key assumptions in our work and justify why they hold well in our setting.

Time synchronization. We assume that the two edge routers $E1$ and $E2$ can be time-synchronized within μseconds, for example, using GPS clocks that many ISPs have already begun to deploy. This is a general requirement for any one-way delay measurement scheme, and in fact is employed by existing edge monitoring solutions such as Corvil [9].

Packet filtering. Two edge routers (say, $E1$ and $E2$) must precisely identify the set of packets over which the metrics need to be computed. However, packets that arrive at $E1$ may not exit via $E2$. We assume some simple way to determine which packets are destined to or from a particular edge router, for example by prefix matching. One could easily construct a simple layer-4 packet filter (using IPs and ports) that clearly specifies the set of packets that travel from $E1$ to $E2$.

No header changes. Measuring latencies would be easy, for instance, if each packet could carry a timestamp that $E1$ embeds with its arrival time that would be subsequently associated with the packet's arrival time at $E2$. However, IP packets have no timestamp field and TCP timestamp options are restricted to carrying *true* end-to-end delays where ends are the actual sockets running on the host machines. Adding a new field is unlikely to happen as it would require intrusive changes to many components in the data path of switches.

C. Issues with earlier solutions

In this part, we illustrate the challenges of fine-grain measurements in a setting with persistent reordering by describing why most relevant earlier solutions do not work well while placing a primary emphasis on the state-of-the-art solution LDA. For providing richer background information, we also briefly summarize other prior solutions in network latency measurements at the end of this section.

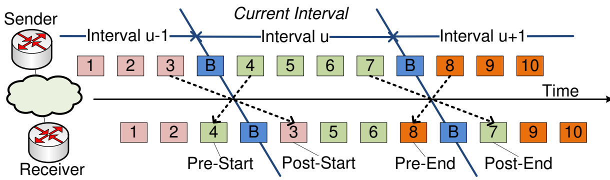


Fig. 2. Four types of reordering that can occur.

Active probes: Injecting active probes (*e.g.*, using ping and other tools such as [10]–[13]) is insufficient due to the requirement for a large number of active probes to measure microsecond latencies (indicated by prior work [4]), interference with regular data packets, and inability to cover all paths between the pairs of edge routers. See §V for a quantitative argument.

Storing timestamps locally: An alternative is to allow the two edge monitors to store packet digests and timestamps locally, and only to exchange them at the end of a measurement interval. However, the storage and communication overhead is extremely high. One could maintain timestamps only for a small *sample* of packets; but, as we show in §V-C, this reduces bandwidth at the cost of accuracy. In a few prior efforts (*e.g.*, [14], [15]), researchers proposed simple router extensions for latency measurements that are similar to this local timestamps idea; and hence they share similar problems.

LDA: LDA suggests a way of greatly increasing the number of latency samples using aggregation. LDA assumes a stream of packets going from a sender Snd (*e.g.*, $E1$ in Figure 1) to a receiver Rcv (*e.g.*, $E2$). LDA starts with the following simple idea: Suppose Snd and Rcv agree upon an interval of T packets in the stream over which they want to measure delay. To do this, Snd marks off intervals by sending a special 'sync' control message each time it sends T packets to Rcv. (Note that Snd could choose to mark the intervals based on time as well.) All packets 'bookended' by a pair of sync messages belong in a single interval. For convenience, we shall refer to the first sync message as an interval-start message, and the next sync message as an interval-end message hereafter.

Snd and Rcv could then compute the average delay by each locally maintaining a sum of packet timestamps (a *timestamp accumulator*) and a count of the number of packets in the interval (a *counter*) which together constitute a *bucket*. The average delay is then the difference between the timestamp accumulator at Snd and timestamp accumulator at Rcv, divided by the number of packets in the counter. However, packet losses between Snd and Rcv can have the bucket updated by *different* sets of packets, thus making latency estimation impossible. To overcome this issue, LDA keeps an array of M buckets and uses packet sampling to maximize useful delay samples. A *crucial assumption* that makes LDA work is that the synchronization messages and packets are delivered in order at the receiver so that the sender and receiver compute delay estimates over the same set of packets; this FIFO assumption makes LDA unsuitable for our setting.

Specifically, Figure 2 shows packets arriving out of order when traversing the network. The ordering of packets that are both transmitted and received within the interval 'bookends' does not affect LDA, since the timestamp accumulators and counters are order-agnostic (addition is commutative). However, there can be the following type of *problematic reordering*, namely packets that start out in one interval at Snd, drift into an another interval at Rcv. This situation is problematic since the timestamp accumulators at Snd and Rcv may be computed based on two different sets of packets, and this difference can affect the delay estimates significantly.

Why does Per-Path LDA not work ? The obvious fix to LDA is to extend it to operate on a per-path basis. All we need then is to average per-path latencies of all the paths between the sender and receiver. Unfortunately, neither the senders nor receivers know which path a given flow will take and so, separation by path is difficult. While we can exploit the fact that ECMP does not reorder TCP flows, LDA for potentially millions of separate TCP flows would pose a scaling problem. One may sample a sufficiently large number of flows to ensure (with high probability) that at least one flow is sampled per path. However, the sampled flows may have too few packets and thus the number of LDA samples can be too small to provide sufficient accuracy. Increasing the number of samples will require either more memory (by sampling more flows) or assuming very skewed distribution of flows (and mechanisms to capture such flows).

In order to address these shortcomings, we propose a new data structure called FineComb, that only keeps storage per destination switch and yet has very high sample efficiency.

Other solutions: Packet Doppler [16] is a passive measurement solution that estimates a delay distribution by observing how packet arrival distributions change as packets traverse from one router to the other. However, its granularity is in the order of milliseconds, not microseconds. In [17], Lee *et al.* describe a per-flow switch-level latency measurement architecture. In our work, we focus on aggregate measurements, so our goals are different from theirs.

III. FineComb

FineComb is designed to work in the measurement model outlined in §II-C, in which problematic reordering can occur at the fringes of a measurement interval. We begin our discussion of FineComb by further articulating the model.

A. Measurement model

There are four types of problematic reordering (see the bottom of Figure 2). First, packets sent at the end of interval $u-1$ can be routed on a high latency path and hence arrive at Rcv *after* the interval-start message. This can pollute interval u with extra packets; we call such packets *post-start* packets. Second, packets from the start of interval u can be routed on a low latency path and hence arrive at Rcv before the interval-start message for interval u, so these *pre-start* packets from interval u are effectively missing. Similarly, packets reordered around the end of the interval were referred to as *post-end* and *pre-end* packets. We say $\rho = R/T$ is the *reordering rate* for interval u, where R is the total number of problematically reordered packets (sum of all the four types). Note that Table I outlines definition of notations used in this paper.

It is crucial to note that R is almost always much smaller than T even if there is persistent reordering. This is because

problematic reordering is confined to the reordering that occurs relative to the interval-start and end messages. Suppose the interval-start and end packets are routed on one path and the rest of the packets are sent on the other path. Thus $R \leq 2CL$, where C is the transmission speed and L is the maximum difference in latencies of paths. For example, if C is 10 Gbps, L = 100 μsecs and an average packet size is 250 bytes, R is around 1,000 packets. If an interval is set as short as 200 μsecs, all 1,000 packets in the interval are problematically reordered. By contrast, T may be as large as 5 million with 1 second interval. Therefore, an interval should be chosen to ensure $R \ll T$.

In addition to reordering, packets can also get dropped in the network, which can cause the Snd and Rcv state to become inconsistent. We assume at most βT packets from interval u will be dropped as they traverse the network from Snd to Rcv, where β is the *loss rate* for interval u.

Now, if we compare the two streams of packets that belong to interval u at the Snd and Rcv sides, the difference between them is at most $\beta T + R$ packets. If we could somehow correct for these $\beta T + R$ *bad packets* that prevent the Snd and Rcv from agreeing, we could make use of the simple timestamp accumulator and counter idea described in §II-C.

B. Key ideas

As in LDA, FineComb keeps an array of M timestamp accumulators and counters at the sender and receiver; a hash function computed over packet contents is used to map each incoming packet to a *bucket* containing a (timestamp accumulator, counter) pair. If the sender and receiver use the same hash function, then they will map packets to buckets in an identical fashion. We say that a *bucket is useful*, if it contains the same set of packets at both the sender and receiver, and thus can be used to compute the delay estimate. Notice that a bucket is useful as long as none of the $\beta T + R$ bad packets hash to that bucket. FineComb corrects for the $\beta T + R$ bad packets using the following three ideas.

1) Incremental stream digests: With reordering, we cannot simply compare counters at sender and receiver and conclude that a bucket is useful; this follows from the fact that a dropped packet that hashes into a bucket can be replaced by a (different) misordered packet from another interval. Even one such event can throw off the delay estimate considerably. The misordered packet may have been sent just before the start of interval u but may hash into the same bucket as a lost packet sent towards the end of interval u. Thus the induced error can be as large as the size of a measurement interval (say 1 second).

To detect such cases, we augment the counter in each bucket with what we call an *incremental stream digest* (ISD). An ISD on a stream of packets pkt_1, ..., pkt_t is computed as follows:

$$H(pkt_1) \odot H(pkt_2) \odot ... \odot H(pkt_t) \qquad (1)$$

where $\odot$ is an invertible commutative operation like XOR, H is a hash function, and $H(pkt_t)$ means a digest. Our incremental packet digests are similar to the incremental collision-free hash functions proposed in cryptography [18]. However, since we are not operating in an adversarial setting, H can be a simpler hash function such as BOB [19] or H3 [20].

The ISD has three useful properties. First, two streams containing different packets will hash to different values with high probability. Second, because $\odot$ is commutative, two streams containing the same set of packets in different order still hash to the same value. Thus we can determine if a bucket is useful by verifying that the ISDs match at the sender and receiver. Finally, we can easily add or subtract packets from the ISD by computing the XORs of their digests with the ISD, which is the basis of *stash recovery* that we describe next.

2) Stash recovery: A stash is simply a set of the timestamps and digests of a small number of packets that arrive before and after the sync messages that delimit an interval. One design option is to maintain one stash at each of the sender and receiver in order to detect which packets are reordered around the boundary of the interval, which negates the necessity of ISD in the bucket. However, the detection is impossible only using the sender and receiver stashes because of the possibility that problematically reordered packets may not be in both stashes as their delays can become much higher than the maximum permissible one-way delay due to network churn, routing loop and misconfiguration. Hence, both the sender and receiver still need to keep one ISD in each bucket, and only the receiver keeps a stash. One nice feature of not keeping a stash at the sender is that if we grow the stash size (especially in a DRAM implementation of the stash), the control bandwidth does not grow with stash size: The sender only needs to send its buckets to the receiver to compute estimates.

As we have seen, this number R is small (say 1,000). Since these are the most likely packets to have been reordered, stash recovery simply attempts to add or subtract the digest of each stashed packet from the corresponding bucket into which that stashed packet hashes. Note that if the stash were as big as T, we would be back to the naive algorithm of storing all local timsetamps. Thus the fact that R is much smaller than T is crucial to the efficiency of stash recovery.

To show a concrete example of stash recovery, suppose a post-start packet P from interval $u-1$ is hashed into the 20th bucket in interval u, making it useless. Assuming P is stored in the stash at the receiver because it arrived shortly after the interval-start message, stash recovery will look up the bucket 20, and try to subtract P's digest from the ISD at the receiver. If the resulting ISD matches the ISD of bucket 20 at the sender, bucket 20 can be made useful again by subtracting the timestamp of P from the receiver timestamp sum. While we have lost 1 sample from the bucket, we have saved perhaps 10,000 remaining samples that aggregate into bucket 20 that would have been lost otherwise. Given memory S, however, it is not clear whether to allocate more stash (and hence, to recover from more reordered packets) or to use more buckets (and hence, to be more resilient to loss); we will investigate this tradeoff analytically and experimentally.

3) Packet sampling: In many practical situations, the number of bad packets $\beta T + R$ is going to be far greater than the number of buckets, M. Given packets are randomly hashed to buckets, that means, that all the M buckets could become useless. Even if somehow, we manage to recover all the reordered packets in a given interval, the number of lost

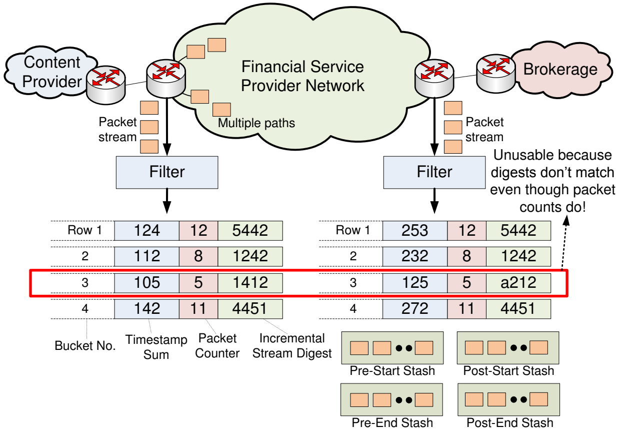


Fig. 3. FineComb. Four stashes cater to the four types of reordered packets.

packets alone βT could be bigger than M. In FineComb, we sample packets at rate p, so that the expected number of bad packets that can make buckets useless drops to $p(\beta T + R)$. On the one hand, selecting a high value of p will mean that the number of bad packets, and in turn useless buckets, will increase. On the other hand, selecting a low value of p will make each bucket aggregate fewer samples. Determining the optimal value of p that maximizes the number of useful samples is a key question that our later analysis will address.

C. Basic FineComb without a stash

We first describe basic FineComb without a stash (as shown in Figure 3) which uses M buckets, each containing a timestamp accumulator, counter and ISD. Each packet is sampled with probability p, and then distributed to one of the M buckets by a hash function. The pseudocode outlined illustrates the steps involved in updating FineComb state at both the sender and receiver for every *sampled* packet. Let $TS[i]$ denote the timestamp accumulator, $C[i]$ the packet counter, $D[i]$ the ISD for ith bucket, and τ the arrival time of packet pkt.

1: $D \leftarrow compute_hash(pkt)$ $\rightarrow$ Digest
2: $i \leftarrow D \ mod \ M$
3: $TS[i] \leftarrow TS[i] + \tau$, $C[i] \leftarrow C[i] + 1$, $D[i] \leftarrow D[i] \odot D$

After sending T packets (or, alternately after a fixed amount of time), the sender sends its set of buckets to the receiver in the sync message. When the receiver receives the sync message, it uses the sender's buckets along with its local buckets to compute the average latency and loss as follows:

1) Estimating average latency: The receiver first determines the set of useful buckets by checking which buckets have matching ISDs at the sender and receiver. For all these 'valid' buckets, the receiver computes the difference between the receiver's and sender's timestamp accumulator, sums them together and divides it by the sum of all packet counters in these valid buckets. The steps are outlined below.

1: $N \leftarrow 0, D \leftarrow 0$
2: **for** i=1, M **do**
3: **if** $C_s[i] = C_r[i]$ and $D_s[i] = D_r[i]$ **then**
4: $D \leftarrow D + (TS_r[i] - TS_s[i])$, $N \leftarrow N + C_r[i]$
5: $Average \ delay = D/N$

While LDA matches the packet counters alone, FineComb does an extra check for a match of the sender and receiver packet digests, which is the main difference between them.

2) Estimating standard deviation: We compute standard deviation in a similar fashion using a technique introduced in [21]. The original technique needs to maintain an additional counter to which each sampled packet's timestamp is added or subtracted with equal probability $1/2$. In order not to waste memory with an extra counter per bucket, we use a trick used in LDA. Note that we leave readers to refer to Section 3.4 in [4] for details on the trick. Checking ISDs of buckets is again required when applying the trick.

3) Loss measurement: Loss measurement becomes difficult in the presence of reordering. While we will go over the reason in a minute, for now, suppose that the effects of packet reordering can be removed somehow. Then, it is easy to see that the following simple algorithm does the job.

1: $N \leftarrow 0, L \leftarrow 0$
2: **for** i=1, M **do**
3: **if** $C_s[i] \geq C_r[i]$ **then**
4: $L \leftarrow L + (C_s[i] - C_r[i])$, $N \leftarrow N + C_s[i]$
5: $loss \ rate = L/N$

Note that the algorithm checks whether a sender counter is greater than the corresponding receiver counter (line 3) because the sender-side counter can be smaller than the receiver-side counter for a particular bucket. Such a situation can easily happen if a few packets drift from the previous interval to the current interval (*i.e.* post-start packets overtaken by the interval-start message). Thus, these packets are not just accounted for in the bucket of the sender in the current interval.

Because the LDA operated in a setting where there was no reordering, it was not designed to exclude the counters in buckets contaminated by the reordered packets, from loss estimation. By contrast, FineComb tries to disentangle reordering from real loss and to achieve higher accuracy using stash recovery outlined in §III-D. But, checking the condition in line 3 is still critical because stash recovery can be imperfect. Further, if a lost packet and a reordered packet that is stored in the stash are *both* hashed to the same bin, stash recovery will fail, because the lost packet has made the bucket 'useless'.

In more detail, assume that before stash recovery $C_s[i]$ for some bucket i was less than $C_r[i]$ because of two post-start packets $P1$ and $P2$ that were hashed into bucket i at the receiver but not at the sender. Suppose further that a third packet $P3$ that hashes into bucket i at the sender is lost on the way to the receiver. Then even if $P1$ and $P2$ are in the stash at the receiver, there is no way for the receiver to correct bucket i because, by definition, it does not have the digest for $P3$ which is lost. Thus, not only is bucket i just useless in calculating delay, the algorithm also cannot tell apart a loss of 1 packet and a reordering of 2 packets in bucket i (as in the example) from a loss of 2 and reordering of 3 packets (say). Thus, the loss estimation algorithm above will ignore bucket i, and thus lose a data point for loss estimation.

Since we try to measure small losses, this is potentially serious. However, with careful sizing of the sampling probability the probability of both a lost packet and a reordered packet hashing into the same bucket is even smaller.

Note that while measuring the variance of loss rate can be of interest, too, it is unclear how to extend FineComb. We leave this as part of future work.

D. Managing the stash

We now describe the details of how to use a stash. Recall that the stash stores individual timestamps and digests for the packets that are most likely to be problematically reordered. The stash only maintained in the receiver, as described in §III-B, is broken up into four *substashes* (pre-start, post-start, pre-end, post-end stash) of size w, where $4w = W$, corresponding to the four types of problematic reordering.

Populating the substashes. Even though the receiver does not know when interval-start message will arrive, the receiver can still populate the pre-start substash as follows. The receiver stores the digest and timestamp in a cyclic queue of length w, such that a new sampled packet causes the oldest packet in the queue to be evicted if the queue is full. The receiver stops populating the stash when the interval-start message arrives. To populate the post-start stash, the receiver keeps a queue of length w that starts being populated once the interval-start message is received, and populates it until it is full. The other two stashes are managed similarly with interval-end message.

Stash recovery. For each useless bucket i, the receiver considers all the entries ($\mathbb{T}$) of the four substashes ($\mathbb{S}$) that map to that bucket. The receiver then considers *all subsets* ($\mathbb{Z}$) of the stash entries that correspond to this bucket. For each subset of stash entries, the receiver XORs the digests of the entries with the bucket's ISD. If the sender's and receiver's ISDs match for this subset of stash entries, then the receiver can recover that bucket by subtracting (if the packet is from the post-start stash or pre-end stash), or adding (otherwise) the timestamps of those stash entries from/to the bucket's timestamp accumulator.

1: $\mathbb{T} \leftarrow build_stash_entry_set_for_bucket(i, \mathbb{S})$
2: **for** all $\mathbb{Z} \subset \mathbb{T}$ **do**
3: $\quad D_r \leftarrow D_r[i]$, $TS_r \leftarrow TS_r[i]$, $C_r \leftarrow C_r[i]$
4: $\quad$ **for** all $(D, \tau, k) \in \mathbb{Z}$ **do**
5: $\quad\quad$ **if** $k =$ pre-start *or* $k =$ post-end **then**
6: $\quad\quad\quad D_r \leftarrow D_r \odot D$, $TS_r \leftarrow TS_r + \tau$, $C_r \leftarrow C_r + 1$
7: $\quad\quad$ **else**
8: $\quad\quad\quad D_r \leftarrow D_r \odot D$, $TS_r \leftarrow TS_r - \tau$, $C_r \leftarrow C_r - 1$
9: $\quad$ **if** $D_s[i] = D_r$ **then**
10: $\quad\quad D_r[i] \leftarrow D_r$, $TS_r[i] \leftarrow TS_r$, $C_r[i] \leftarrow C_r$, return

Stash recovery appears to take exponential time because it may seem that one has to consider all possible combinations (2^W) in the worst case when W stash packets hash to a single bucket. Fortunately, stash recovery is much faster because, with high probability, only $O(W/M)$ stash packets can hash together into the same bucket. Thus, the running time of the decoding algorithm is $O(M2^{W/M})$, and since the typically stash size $W < M$ number of buckets, it follows that stash recovery time is approximately linear in M.

Thus the algorithms to calculate loss and latency are exactly as before for basic FineComb except that we preface them by doing stash recovery to potentially increase the number of useful buckets. A stash should help improve latency estimates slightly (by increasing the number of useful buckets), but will be much more critical in obtaining reasonable loss estimates (allowing loss to be distinguished from reordering).

Symbol	Meaning
T	The number of packets sent per interval
ρ	The reordering rate in an interval
R	The number of problematically reordered packets in the interval ($= \rho T$)
β	The fraction of dropped packets in the interval
G	The number of *good packets* that are received by the Receiver between the correct pair of 'Sync' messages
p	The FineComb sampling rate
M	The number of buckets in the FineComb
W	The number of entries in the stash
S	The total storage allocated to the FineComb with stashes ($= M + W$)
X	A random variable describing the number of useful samples extracted from FineComb
L	A random variable describing the number of *bad packets* that are *sampled* and *not corrected* during stash recovery

TABLE I
NOTATIONS.

E. Handling unknown loss and reordering rates

If we know the exact reordering rate ρ and loss rate β a priori, our theoretical results (shown in §IV) allow us to configure the sampling rate optimally. In practice, however, we do not know these values a priori and they may change over time. LDA also faces a similar problem with unknown loss rate, and hence it maintains multiple banks each tuned to different loss rates. We can use a similar trick in FineComb as well, except, we need to consider the operating ranges of two different parameters β and ρ. We use multiple banks optimized for the four operating regions: $(\beta_{min}, \rho_{min})$, $(\beta_{min}, \rho_{max})$, $(\beta_{max}, \rho_{min})$, and $(\beta_{max}, \rho_{max})$ because it can cover varying loss and reordering rate in a balanced way. Low values of β_{min} and ρ_{min}, mean that the sampling rate chosen could be high, which in turn means the estimates are good. Once the loss rate or reordering rate becomes high, this bank tuned for low loss and reordering rates may produce no valid delay or loss estimates. While we believe other configurations (*e.g.*, 6 banks) would work fine, we have not explored them yet.

In 4-bank FineComb, each uses one fifth of the total storage. We compute the optimal sampling probabilities and stash size for each operating region independently and partition resources statically. Each bank has a different number of buckets from each other. We then make the number of buckets of all banks equal using the remaining one fifth of the total storage unused.

For estimating delay, we take the maximum count among counts from four buckets in the same row (the same index) across banks and its corresponding timestamp sum, and add each values with a total count and a total timestamp sum, respectively. We repeat this step for all rows. This procedure provides the maximum total number of samples. For loss estimation, we pick the loss rate of a bank whose estimate is closest to what it was tuned for. We observe this heuristic works well in our experiments. Further refinement may be viable by using estimates from other banks, but we have not explored it yet.

IV. SETTING PARAMETERS

In the following analysis, our goal is to choose a sampling rate p and stash size W that will maximize $E[X]$, the expected number of delay samples that we extract from FineComb. That is, we would like to maximize the expected number of packets

that are hashed to useful buckets, so that we can estimate delay as accurately as possible. The following analysis assumes that FineComb uses a single sampling rate p, and that the number of entries in the stash and the number of buckets in FineComb M is fixed, so that total storage is $S = M + W$.[2] Note that while we only state the main theorems, results and proof sketches here, refer to the Appendix for additional proofs.

A. Expected number of useful samples

Since our goal is to maximize $E[X]$, our first step will be to determine $E[X]$.

Good and bad packets. Let us focus on interval u, and say a packet sent by the sender in interval u is 'good' if it was received by the receiver within the boundaries of interval u (see §II-C or Figure 2), otherwise 'bad'. Recalling that β is the packet loss rate on the path, T is the number of packets the sender sends in an interval, the number of good packets is $G \le (1-\beta)T$ with equality when $R = 0$, so that there are no packets that are problematically reordered. Packets can become bad due to loss or problematic reordering. The number of dropped and reordered packets in an interval is βT and $R = \rho T$ respectively.

Conditional expectation of useful samples. Let L be the number of bad packets that are sampled but not corrected during the stash recovery. We can prove that the expected number of useful samples is

$$\begin{aligned} E[X|L] &= E[\text{Good pkts per bucket}]E[\text{No. of useful buckets}] \\ &= \tfrac{p}{M}G \cdot (M - E[K|L]) = pG(1-\tfrac{1}{M})^L \end{aligned} \quad (2)$$

where, following [22], we let K be a random variable that denotes the number of 'useless' buckets in the LDA, that results from the L *sampled* bad packets hashing to buckets of the LDA. In [22], they show that K is distributed as

$$\Pr[K = k|L] = \frac{M!}{(M-k)!}\frac{S(L,k)}{M^L} \quad (3)$$

where $S(L,k)$ is a Stirling number of the Second Kind. Using (3), we obtain $E[K|L] = M(1-(1-\frac{1}{M})^L)$ (proof in Claim A.1 in the Appendix) so that (2) follows by substitution.

Sampled uncorrected bad packets, L. We have βT dropped packets and R reordered packets; together, this gives us $\beta T + R$ bad packets, that we sample with rate p. We shall assume that *every* packet that is stored in the stash is an out-of-order packet, so the stashes will allow us to correct for exactly W sampled out-of-order packets. (We make this assumption because it is hard to predict the distribution of problematically-reordered packets. Indeed, in practice we expect the stash to store some packets that arrived correctly in an interval (these good packets waste space in the stash), as well as some out-of-order packets. Thus, our analysis will size the stash under the assumption that the stash does the 'best it can' to correct for reordering.) Thus, the expected number of bad packets that are sampled and not corrected is

$$E[L] = \beta p T + \max\{0, pR - W\} \quad (4)$$

[2]We could instead fix the total storage of the system, so that $S = 2M + W$, since the sender has no stashes and thus requires storage M, while the receiver requires $M + W$ storage.

The distribution of L is evaluated in §A in the Appendix.

Working with the conditional expectation. Because the distribution of L is quite complicated, (see §B in the Appendix), in this section, we work with the conditional expectation $E[X|L = E[L]]$, which is obtained by plugging (4) into (2). By numerically plotting equations, we observed the results obtained using $E[X|L = E[L]]$ are quite close to results obtained from the unconditional distribution $E[X]$.

B. Optimizing stash W for fixed sampling p

First, we would like to optimize the ratio between the bucket size and the stash size to maximize $E[X]$. To do this, we plug (4) into (2) and use the fact that $S = M + W$. We observe that there are two regimes for which the stash size W maximizes $E[X|L = E[L]]$:

$$W \approx \begin{cases} pR & \text{when } S \ge p(R + \beta T) \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

In §C in the Appendix, we show that this holds even when we work with $E[X]$ (rather than just $E[X|L = E[L]]$).

Notice that (5) suggests that when the total storage S is very small, *i.e.* less than the number of bad sampled packets, all the storage should be dedicated to the buckets of FineComb (*i.e.*, W=0). On the other hand, when we have a decent amount of storage, the analysis shows that we should keep stashes large enough to correct for the expected number of out-of-order sampled packets, pR. This makes sense, since a single bad packet can cause an entire bucket to become useless, so that about $\frac{p}{M}G$ 'good' packets become useless. Hence, it follows that correcting a single discrepancy in FineComb due to a bad packet is highly effective, and further that we should dedicate a large amount of storage to the stash.

C. Optimizing sampling rate p.

No stash. Per (5) we now consider the case where we have no stash. We can show that the optimal sampling rate is

$$p^{**} = \min\left\{\frac{S}{R + \beta T}, 1\right\} \quad (6)$$

To obtain (6), we use the fact that $E[X|W = 0]$ is easy to obtain in closed form from (2) by observing that L is a binomial random variable with mean $p(\beta T + R)$. Approximating L as a Poisson random variable, and putting $M = S$, using (2) we have that

$$\begin{aligned} E[X|W=0] &= E[X|L]\Pr[L = \ell] \\ &= \sum_{\ell=0}^{\infty} pG(1-\tfrac{1}{S})^{\ell} \cdot e^{-p(\beta T + R)}\tfrac{p(\beta T + R)^{\ell}}{\ell!} \\ &= pGe^{-p(R+\beta T)/S} \end{aligned} \quad (7)$$

The claim follows by taking the derivative of $E[X|W = 0]$ and setting it equal to zero.

Stash. Now, (5) tells us that when we have a stash, its optimal size is $W^* = pR$. We can show that when we use this value for the stash, the optimal sampling rate is approximately

$$p^* = \min\left\{\frac{S}{2\rho^2 T}\left(2\rho + \beta - \sqrt{4\rho\beta + \beta^2}\right), 1\right\} \quad (8)$$

where $\rho = R/T$. We obtained this value by setting $W^* = pR$ and $M = S - W^*$ to obtain $E[X|L = E[L], W = W^*]$ from (2) and (4). We then find p^* as the value that maximizes $E[X|L = E[L], W = W^*]$ by taking its derivative and setting it equal to zero. In §D in the Appendix, we show this value of p^* also (approximately) maximizes $E[X|W = W^*]$.

To stash or not to stash. The last issue we need to settle is whether it is better to use a stash or not. Plugging our two operating points (p^{**}, $W = 0$) and (p^*, $W = p^*R$) into the equation for $E[X]$, we find (see §D in the Appendix) that $E[X]$ is maximized when we use a stash.

A note on our approach. This analysis first fixed the sampling rate p and then optimized stash size W; then the optimal value for W was used to solve for the optimal sampling rate p. It would have been better to jointly optimize $E[X]$ for W and p; however, the complexity of $E[X]$ made a joint optimization quite complicated, so we leave this as future work.

V. Evaluation

In this section, we evaluate the efficacy of FineComb. Specifically, we seek to answer the following questions: (1) What is the relative error of FineComb in estimating mean delay, standard deviation and loss rates under different levels of reordering and loss rates? (2) How does an optimal configuration of FineComb compare with previous solutions assuming the same total memory for given loss and reordering rates? (3) Since loss (β) and reordering (ρ) rates are not known a priori, we evaluate the efficacy of the multi-bank FineComb that is tuned towards different β and ρ values. Before we answer these, we first describe our evaluation methodology.

A. Evaluation methodology

We built a custom simulator in C++ for evaluation. Our custom simulator is more efficient than, say, ns-2 and allows us to simulate sending several million packets. Further, ns-2 does not provide any built-in routines that we can leverage as all we need is to simulate packets sent on a link with specified delay, loss, and reordering characteristics.

Given our goal is to compare the performance of our architecture in many different settings, we provide several configuration parameters such as loss rate β, reordering rate ρ and measurement interval. Our simulation environment is deliberately kept similar to the one used by the authors in [4] so that fair comparison of FineComb with LDA is possible.

Delay model. Ideally, we would use traces at two monitoring endpoints within a real data center with GPS synchronized clocks to estimate end-to-end latency; unfortunately, there exist no such publicly available data center latency traces. Prior work [4] used the Weibull delay distribution model empirically verified to mimic the distribution of delays within a backbone router by Papagiannaki *et al.* in [23]. The delay for each packet is drawn from a Weibull distribution, which has cumulative distribution function $P(X \leq x) = 1 - e^{-(x/\lambda)^k}$ with k and λ denoting the shape and scale of the graph respectively. While following shape parameter $0.6 \leq k \leq 0.8$ recommended in [23], we choose $k = 0.6$ in all our simulations. We then adjust λ to obtain a mean delay of 10 μs. While we mainly rely on Weibull distribution within our simulations, we use a real trace collected from an ingress and an egress interface of a router connected to an OC-3 link (155 Mbps) to evaluate the multibank scenario (refer to §V-D for more detail). Note that while FineComb and LDA are agnostic to the distribution of timestamps, delay distribution does matter when we determine the relative error provided by these data structures.

Loss model. FineComb and LDA are agnostic to the loss rate distribution—even if two lost packets are back-to-back, they are randomly hashed into different buckets anyway. Thus, it suffices to simulate *random* packet loss.

Measurement interval. We simulate an interval of 1 second with a mean delay of 10 μs. (Path latencies in data centers may range from 10–100 μs, so our setting simulates close to the finest granularity.) Our results are presented as relative error, so exact delay average does not matter. We simulate 5 million packets, with an average packet size of 250 bytes (similar to [4]), over a 10 Gbps bottleneck capacity with an inter-arrival time of 0.2 μs—transmission time for 250 bytes at 10 Gbps. All our simulation results are an average across 10 runs.

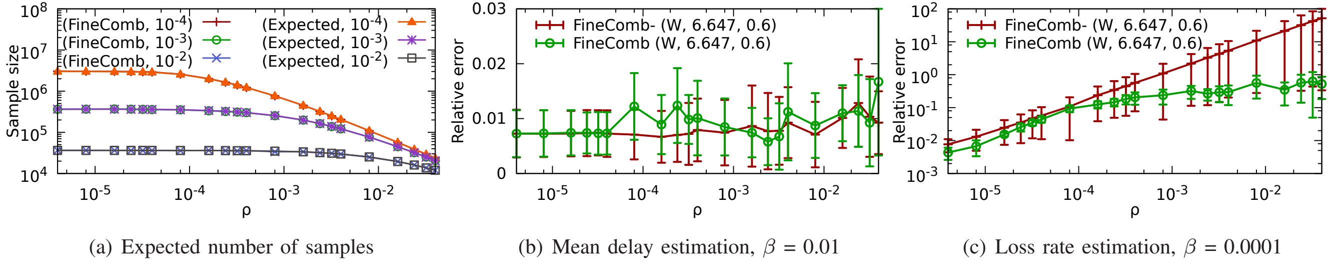
Reordering model. An important parameter is the reordering rate ρ. We could simulate reordering in the same way we simulate loss; by randomly choosing which packets to reorder. However, in practice, it is not at all clear that reordering follows a similar process of loss; in fact, there is no generative model that we are aware of that we can use in our simulation. We note once again that reordering within the interval affects neither LDA nor FineComb; what matters is problematic reordering at the fringe of an interval (see Figure 2).

To stress LDA and FineComb in terms of problematic reordering, we simulate the following simple deterministic model of reordering. In our reordering model, we essentially specify a 4-tuple, $<R^s_{pre}, R^s_{post}, R^e_{pre}, R^e_{post}>$, the number of pre-start, post-start, pre-end and post-end packets defined in §III-A. Then, for each interval we wish to simulate, we choose a contiguous set of packets from the end of one interval that will drift into the next and vice-versa.

Note that the theory in §IV is based on the total number of reordered packets $R = \rho T$ and considers a slightly more simplistic model than we use in our experimentation. While clearly, $R = R^s_{pre} + R^s_{post} + R^e_{pre} + R^e_{post}$, the optimal probability p^* obtained in (8) is computed assuming all these different individual reordering components are the same. To make our provisioning strategy consistent with the theory, we obtain the total reordered number of packets R as follows:

$$R = max\{R^s_{pre}, R^s_{post}, R^e_{pre}, R^e_{post}\} \times 4$$

We simulate two main types of reordering, called *forward* and *backward*, that correspond to $<0, x, 0, 0>$ and $<x, 0, 0, 0>$ configurations for the 4-tuple. In most experiments, we configure x equal to roughly $10^{-6}T$ to $10^{-2}T$; equivalently, the reordering rate ρ varies from $4 \cdot 10^{-6}$ to $4 \cdot 10^{-2}$, translating to roughly 50 to 50,000 packets. We also simulated many other configurations (*e.g.*, $<x, x, x, x>$, $<x, x, 0, 0>$) but latency estimation results were mostly similar in all cases; this follows because sampling probabilities and stash sizes are all dependent on ρ, which is same for all these configurations.



(a) Expected number of samples (b) Mean delay estimation, β = 0.01 (c) Loss rate estimation, β = 0.0001

Fig. 4. Expected number of samples obtained by FineComb, and relative error of mean delay and loss estimates in the presence of backward reordering under Weibull distribution. Error bars represent standard deviations of estimation errors. We show both FineComb and FineComb- for comparison.

Resource configuration. We allocate a total of 1,000 buckets for FineComb. To simulate cases with and without stash, we assume stash elements are of the same size as bank elements (for simplicity). We use 64 bits from a 160-bit SHA-1 hash function for packet digests. To make things fair, we equalize the storage at the LDA and the FineComb. The buckets in the LDA are 2/3 the size of those in FineComb (LDA has timestamp accumulator and counter but no ISD). Furthermore, while FineComb is asymmetric (only the receiver maintains stashes), the LDA is symmetric. Thus, LDA gets $1.5(M + W/2)$ buckets at sender and receiver.

B. Assessing FineComb

Expected number of samples. Our first experiment aims to understand how tight the theoretical bound on the number of useful samples is, at the optimal sampling probability. Figure 4(a) shows the expected number of samples according to the analytical bound given in (2) (curve titled 'Expected') and the empirical number of samples over which delays are computed. The three different curves in the figure correspond to three different loss rate settings (10^{-4}, 10^{-3}, 10^{-2}). Clearly, as the loss rate increases, the number of effective samples reduces all the way from almost 3 million packets at loss rate 10^{-4}, to about 40,000 packets at loss rate 10^{-2}. As we increase the reordering rate, the number of effective samples also decreases (although not by much for the 0.01 loss rate curve, since the loss rate overwhelms the reordering rate significantly). This is expected since more losses render more buckets useless, which in turn decreases the expected number of samples. In all cases, we observe that the analytically expected number of samples matches quite well with what we found empirically (the curves are virtually indistinguishable); the difference is of the order of a few hundreds.

Latency estimates. Next, we show the average relative error of mean delay and loss estimates, as we vary the reordering rate ρ in Figure 4. We show the results comparing FineComb and FineComb- (FineComb without the stash) for Weibull distribution with parameters adjusted to ensure similar mean latency of 10 μs. While we have simulated many different levels of loss and types of reordering, for brevity, we mainly show the latency results for the high loss situation and loss estimation for the low loss situation. (These are the least favorable situations for FineComb.) From Figure 4(b), we see that the relative error for FineComb is less than 1.2% under different levels of reordering. While we omit the exact figure of standard deviation estimation for brevity, FineComb and FineComb- achieve similar average relative error—less than 9% across all ρ values.

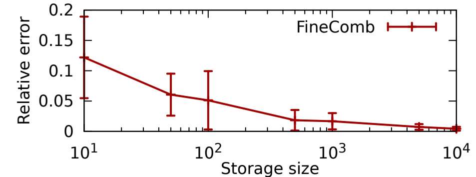


Fig. 5. Impact of storage sizes on mean delay estimation. β = 0.01, ρ = 0.04. Error bars represent standard deviations of estimation errors.

As predicted by our analytical work (see Figure 10 in the Appendix), FineComb provides about 15-30% more useful samples than FineComb-. While more samples should lead to better delay estimates, the improvement in the delay estimate depends heavily on the specific delay distribution; that is, some distributions require fewer samples to obtain accurate estimates (*e.g.*, in an extreme case, a uniform distribution requires only a small number of samples for excellent accuracy).

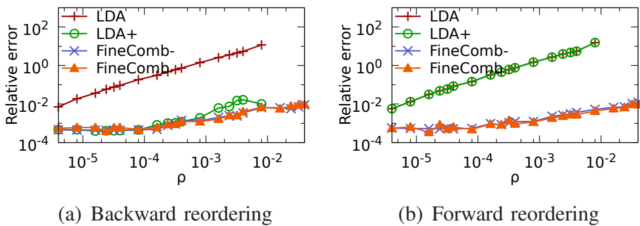
Loss rate estimates. We clearly see the benefit of the stash when we consider loss estimation error in Figure 4(c). We can observe that the estimates of FineComb- are significantly worse than FineComb, especially at higher reordering rates. This is explained by the fact that loss rate estimates for FineComb- include reordered packets; because FineComb- has no stash, we have no way to prevent these reordered packets from polluting our loss rate estimator. Having the stash helps recover most of those reordered packets in FineComb, thus adding significantly fewer number of false positives in calculating the loss rate.

Impact of storage sizes. Given 5 million packets in an interval, we vary storage sizes and study their influences on mean delay estimation. β and ρ are set to 0.01 and 0.04 respectively, which lets FineComb face the most challenging situation. As we expected, estimation error reduces from 12% to almost 0.4% as storage size increases (see Figure 5). Considering the trade-off between storage size and estimation accuracy, we use 1,000 buckets for the rest of the experiments.

C. Comparison with other solutions

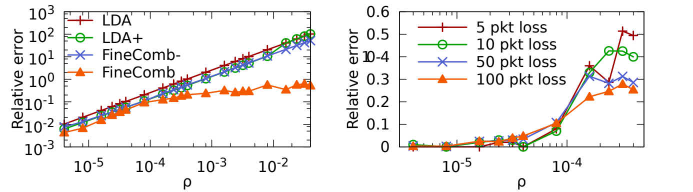
We compare FineComb with LDA using simulations. Before we show these results, however, we go over why other simple alternatives do not work as well as compared to FineComb.

1) Active probing: Intuitively, active probing methods do much worse than methods like FineComb in terms of standard error for a fixed control bandwidth, because each active probe



(a) Backward reordering (b) Forward reordering

Fig. 6. Average relative error of mean delay estimates. β = 0.0001.



(a) Low loss, β = 0.0001 (b) Microscopic loss, 5-100 lost packets

Fig. 7. Relative error of FineComb at detecting low to microscopic losses.

provides a single delay sample, while each FineComb bucket provides thousands of samples. Using a sampling probability of $p = 0.1$ (optimal for low loss and small amount of reordering), FineComb will provide 500,000 delay samples in each interval. Now the control bandwidth required to send 1,000 buckets to the receiver, is roughly 16,000 bytes (assuming 16 bytes per bucket) while an active probe takes at least 64 bytes (packet headers plus timestamp). To keep the control bandwidth the same, even if we allowed $16,000/64 = 250$ active probes per second, they would only provide 250 delay samples while FineComb provides 500,000. This 2,000x increase in sample size translates roughly to $\sqrt{2000} = 44$x decrease in standard error. In addition, probe packets perturb traffic of regular packets, thus distorting true delay in a case of no probes, and without careful choice of probing techniques, it is also hard to achieve unbiased estimation [24].

2) Sampled local timestamps: Similarly, consider the other trivial solution of sampling a small number of packets in each interval and storing their timestamps.

We compare this trivial solution to FineComb- (note from §IV that adding the stash only *increases* the number of good samples). Combining (7) and (8), we find that when FineComb has S buckets and no stash, it produces $E[X_{FineComb}] = S\frac{G}{\beta T+R}e^{-1}$ good timestamp samples. Meanwhile, the trivial solution that samples at rate p obtains $p(1-\beta)T$ good samples while storing $S_{sample} = pT$ items. Setting $S_{sample} = S$, we find that FineComb produces about $\frac{G}{\beta T+R}$ more good samples than the trivial solution; note that we expect this ratio to be much larger than one, since G is the number of 'good' packets, while $\beta T + R$ is the number of 'bad' packets in the interval.

For example, assume that FineComb uses 1,000 buckets and a stash of the same size. Then the trivial algorithm can afford to store 2,000 samples. Once again, for the same parameters as the example above, the trivial algorithm will provide 2,000 samples per second, while FineComb will provide 500,000. This factor of 250x increase in sample size translates to roughly a factor of 15x decrease in standard error.

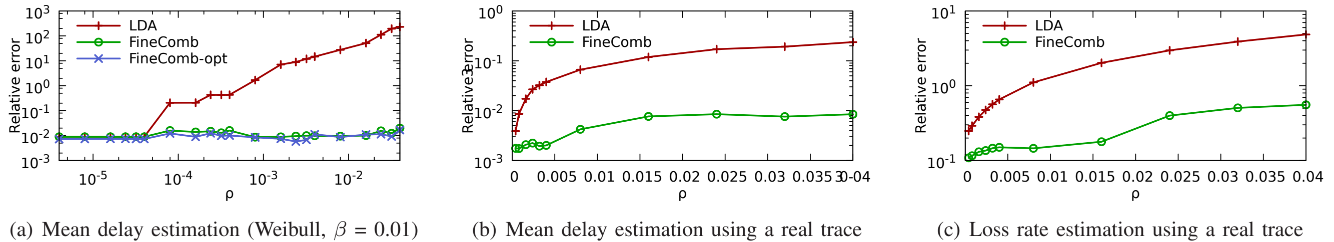
3) LDA for latency estimates: We conduct experiments to know the relative error of mean delay estimates for four solutions, namely LDA, LDA+, FineComb and FineComb- for different reordering rates and reordering models. LDA+ is a simple refinement of LDA. It effectively ignores the set of buckets where the sender timestamp sum is larger than the receiver timestamp sum, which results in *a negative delay* contributed by that bucket. For this set of experiments, we choose the optimal stash size configurations and sampling probabilities (for LDA, as recommended in [4]) for all solutions.

Figure 6 shows that, beyond small levels of reordering, LDA consistently performs worst, with relative error as high as 100% ($\rho = 0.0005$) to 400% ($\rho = 0.004$). This follows from the fact that LDA cannot deal with reordered packets. If a reordered packet and a lost packet hash into the same LDA bucket, the LDA will assume that bucket is useful and include it in the latency estimation. However, that bucket will contain timestamps relating to two *different* sets of packets, and error induced can be as large as the measurement interval (*e.g.*, 1s).

The fact that LDA+ can ignore buckets of contributing negative delays clearly helps solve most of the problems in the backward reordering case (where extra packets drift out of the interval), as reflected in the better relative error for LDA+ in Figure 6(a). In fact, in cases where LDA+ was optimized for higher loss rate (*e.g.*, at $\beta = 0.001$), we observed better accuracy than FineComb, that can be explained by the fact that the total number of buckets allocated to LDA is about 1.5 times higher than those allocated to FineComb, resulting in slightly better sampling rate, and consequently, in more samples. However, LDA+ is merely a patch, and does not work in the forward reordering case, since we cannot easily detect (using a simple elimination scheme as before) and eliminate buckets that are anomalous. This is because although the packet sets in the sender and receiver buckets do not match due to problematic reordering and loss, the counts in the buckets can be the same and the value obtained from the subtraction of timestamp sums is still positive. Thus in Figure 6(b), we see that LDA+ has the same accuracy level as LDA. Note that in Figure 6 both LDA and LDA+ have no useful bucket and in turn obtain no latency estimate when $\rho \geq 0.016$.

In all cases, we observe that both FineComb and FineComb- perform consistently better than LDA even under high loss and reordering rates. The relative error is mostly around 0.1% and never more than 1% in all the cases considered. For standard deviation estimates, we observed a similar phenomenon, *i.e.*, the accuracy of FineComb is orders of magnitude higher than LDA's. The same set of reasons for the case of mean delay estimates explains why standard deviation estimates are also bad. (Since the curves look exactly the same as those for mean latency, we omit them.)

4) LDA for loss estimation: In Figure 7(a), we plot the relative error in estimating loss rate (for $\beta = 0.0001$). FineComb's estimates are usually within 10-30% error irrespective of the reordering rates. The estimates of the rest are quite poor, with more than 100-500% error for LDA. This is expected, since neither LDA (or LDA+) nor FineComb- have the capability to correct for reordered packets; only FineComb enjoys that capability due to the presence of the stash.



(a) Mean delay estimation (Weibull, β = 0.01)
(b) Mean delay estimation using a real trace
(c) Loss rate estimation using a real trace

Fig. 8. Average relative error of mean delay and loss estimates of the 4-bank FineComb, with the banks optimized for low and high reordering and loss rates under backward reordering scenario.

Microscopic losses. While 10-30% error in estimating loss rates as low as 0.0001 is good, our goal was also to detect losses as low as 1 in 1 million (10^{-6}). Intuitively, detecting such low loss rates in the presence of reasonable levels of reordering (*e.g.*, 500 packets, *i.e.*, $\rho = 10^{-4}$) is possible only with extremely high sampling rates (close to 1) and with a stash large enough to recover most of the reordered packets. (Our formulae predict these configurations as well.) To explore this case further, we simulate low loss conditions (with 5, 10, 50, and 100 packets lost in the interval) and configure stash and sampling optimally just as before. The 5 packet situation is equivalent to $\beta = 10^{-6}$. In Figure 7(b), we see that, even though the relative error of FineComb's loss estimates becomes progressively worse as reordering increases, the estimates are well within 10% for reordering rates up to 10^{-4}, *i.e.*, 5 packets lost is reported as either 4 or 6 packets lost—we believe most managers would find such accuracy for microscopic losses to be perfectly adequate. By contrast, LDA's accuracy for the same range is around 2,000% (not shown in the figure), which can cause false alarms.

D. Handling unknown loss and reordering rates

We have already shown that it is easy to tune FineComb if the manager knows the loss and reordering rate. However, it is important to have a solution that works across a large range of loss rates and reordering rates using multi-bank FineComb.

We use Weibull distribution and a real trace to compare the efficacy of 4-bank FineComb with a two-bank LDA under unknown loss and reordering rates. First, for Weibull delay distribution, average latency is set to 10 μs and β is set to 0.01. Second, the trace collected by the authors in [12] contains about 2.4 million packets with real timestamps and true delays over about 150 second interval which experienced queuing delay, packet loss, and so on in a router. The average loss rate of the trace is about 0.24%, but each measurement interval containing 0.2 million packets has different packet loss rates; minimum, median and maximum loss rates are 0%, 0.06% and 0.96%, respectively. There is no packet reordering in the trace. Hence, we simulate reordering rates from 0.0004 to 0.04.

For 4-bank FineComb, we optimize the individual banks for the four pair-wise combinations of $\beta_{min} = 0.0001$, $\rho_{min} = 0.0004$, $\beta_{max} = 0.01$, and $\rho_{max} = 0.04$. Two-bank LDA is optimized for $\beta_{min} = 0.0001$ and $\beta_{max} = 0.01$.

Figure 8(a) shows the relative error of the mean delay estimates of FineComb compared to that of LDA. FineComb-OPT, shown for reference, is FineComb configured with the theoretically best sampling rate and stash size based on the given loss and reordering rates. The results for other loss rates and for the forward reordering case, follow a similar trend shown in the figure that FineComb-OPT works best, FineComb next and LDA worst (and hence, omitted). As shown in the figure, the estimates of LDA become quickly unusable with small increases in reordering rates (at around 0.0002). Further, we can clearly see that, while 4-bank FineComb appears to have slightly worse relative error than the FineComb-OPT, on the whole, FineComb achieves a relative error of less than 1% under almost all conditions. Standard deviation estimation also shows a similar trend with mean estimation, so we omit the exact graph for brevity. As a summary, FineComb has at least two orders of magnitude less errors than LDA when $\rho \geq 0.0008$, and about 11-13% relative errors are obtained by FineComb across all reordering rates.

In Figure 8(b), we observe the similar pattern shown in Figure 8(a). However, compared to the results of Figure 8(a), the degree of inaccuracy of LDA is lower. This may be because of two reasons. First, there are four measurement intervals which have no packet loss. For those intervals, latency estimates were quite accurate because two-bank LDA could absorb the impact of reordered packets considered as lost packets. Second, true average latencies are quite high, ranging from a few to tens of milliseconds. Thus, denominator in relative error is also high and the relative error is small. Nevertheless, compared to FineComb, at least an order of magnitude higher relative error is observed when $\rho \geq 0.0024$. Again, FineComb achieves a relative error of less than 1% under all conditions. Similarly, for standard deviation estimation, LDA showed an order of magnitude higher relative error than FineComb.

In Figure 8(c), we show the relative error of the loss rate estimation. FineComb's relative error is less than 20% up to the reordering rate of 0.016. Beyond that rate, the relative error of FineComb is comparatively worse at around 55%, but LDA is completely unusable across almost all the rates.

VI. Implementation

Implementing FineComb would still be feasible although a high speed SRAM is required to keep up with high line rates and the total memory required in an endpoint is proportional to the number of its associated endpoints. For instance, a data center network architected as a fat-tree topology with 48-port switches can host 27,648 servers. For end-to-end measurements between edge switches, an edge switch needs to maintain 1,152 FineComb data structures, each of which has total 2,000 buckets and stash entries altogether (we have to consider bi-directions; thus, half of the buckets are used for

a sender (one direction) and the other half for a receiver (the other direction)). Assuming a bucket (and a stash entry, too) takes 16 bytes, 36 MB of SRAM would be required. While this constraint sounds prohibitive for deployment of FineComb, we only need four 9 MB of SRAMs; one costs around US $100 and its footprint is 15mm×13mm [25].

Stash recovery operations are easier to do in software using say an on-board processor. In the analysis, we argued that stash recovery times are $O(M2^{W/M})$. We did measurements to verify that the apparent exponential is not an issue, and that there are no large constants hiding behind the order notation. The table below shows stash recovery times for different stash sizes, assuming a fixed total storage S of 2,000 (across sender and receiver). For example, when the stash (maintained at receiver) W is 838, M is 581 (equal across sender and receiver), resulting in $2^{W/M}$ being less than 4. The implementation was done using a 2.33GHz single core Intel processor with running Linux.

Stash size	20	120	200	462	703	838
Time (ms)	1	4	6	10	10	14

As we expect, stash recovery time increases as stash size increases. However, even for a ratio of stash to buckets of 1.44, recovery takes no more than 14 ms. Note that it is not required that the processor be on-board. While packet processing needs to be done on board, functions such as stash recovery can be implemented in software on the PC. Implementing FineComb on boards (based on FPGAs costing a few thousand dollars) is significantly cheaper compared to existing diagnosis boxes proposed for data centers such as those supplied by Corvil. The high-end Corvil boxes costs UK£90,000 for a 2×10 Gbps box [9]. The high cost is a barrier for most data centers, which explains why Corvil has mostly marketed to a niche market (financial traders) where money is no object.

VII. Conclusions

Measurement tools are badly needed to determine fine-grain latencies and losses that can affect application SLAs in data centers environments. Existing scalable approaches such as LDA designed for switch-level measurements work poorly for end-to-end measurements in the presence of packet reordering which actually happens in IP networks. We described FineComb, a simple yet scalable data structure that can detect microsecond latency violations and microscopic losses (as small as few packets in a million) while still being resilient to reordering. FineComb uses two new ideas—the addition of an incremental stream digest to detect mismatches in packet sets, and a simple stash to correct reordering. Stashes are especially powerful to measure microscopic losses accurately. While FineComb is useful for end-to-end measurements in the short-term, we believe that the future will see the rise of reordering tolerant transport protocols in the data center together with packet-by-packet load balancing within and across routers. In such cases, reordering becomes a fact of life and solutions such as FineComb will become essential to measure fine-grain delays and losses even *within* routers.

Appendix

This appendix serves as a companion to §IV. We start with a simple claim:

Claim A.1: If K is as in (3), then $E[K|L] = M(1-(1-\frac{1}{M})^L)$.

Proof: Our proof uses the following identities of the Stirling number of the second kind:

$$S(L,k)\cdot k = S(L+1,k) - S(L,k-1) \quad (9)$$

$$\sum_{k=0}^{L} \tfrac{M!}{(M-k)!} S(L,k) = M^L \quad (10)$$

$$S(L,L)=1,\ S(L,0)=0 \quad (11)$$

And now we begin:

$$E[K|L] = \sum_{k=1}^{L} k \tfrac{M!}{(M-k)!} \tfrac{S(L,k)}{M^L}$$
$$= \tfrac{1}{M^L} \sum_{k=1}^{L} \tfrac{M!}{(M-k)!} (S(L+1,k) - S(L,k-1)) \quad (12)$$

where we used (9). Now, using Equations (10-11), we can find the first term of the sum as

$$\sum_{k=1}^{L} \tfrac{M!}{(M-k)!} S(L+1,k) = \sum_{k=1}^{L+1} \tfrac{M!}{(M-k)!} S(L+1,k) - \tfrac{M!}{(M-L-1)!} S(L+1,L+1) = M^{L+1} - \tfrac{M!}{(M-L-1)!}$$

and the second term of the sum as

$$\sum_{k=1}^{L} \tfrac{M!}{(M-k)!} S(L,k-1) = M \sum_{j=0}^{L-1} \tfrac{(M-1)!}{(M-1-j)!} S(L,j) = M((M-1)^L - \tfrac{(M-1)!}{(M-L-1)!})$$

and plugging these back into (12) we get

$$E[K|L] = \tfrac{1}{M^L}(M^{L+1} - M(M-1)^L) = M(1-(1-\tfrac{1}{M})^L)$$

as required. ■

A. Distribution of L

We have βT dropped packets, and R reordered packets; together, this gives us $\beta T + R$ bad packets, that we sample with rate p. It is exactly these bad packets that can cause certain buckets of the FineComb to become useless. If we assume that the stashes can correct for at most W bad out-of-ordered sampled packets, the expected number of bad packets that are *sampled* is a random variable $L = A + B$ where A is number of *sampled* out-of-order packets in the interval that are *not corrected during stash recovery*, and B is the number of *sampled* dropped packets in the interval. Notice that A and B are independent random variables, where B is a binomial random variable $B \sim \mathcal{B}(\beta T, p)$, and A is distributed as

$$\Pr[A=a] = \begin{cases} \sum_{i=0}^{W} \binom{R}{i} p^i (1-p)^{R-i} & \text{when } a=0 \\ \binom{R}{a+W} p^{a+W} (1-p)^{R-a-W} & \text{when } a \geq 1 \end{cases}$$

Since $L = A + B$, we can find the expectation of L as in (4).

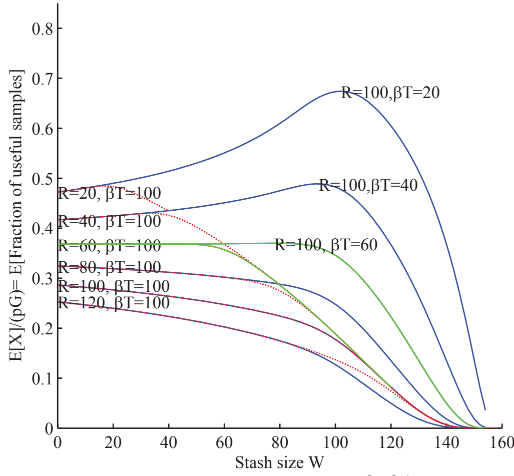


Fig. 9. Expected fraction of useful samples $E[X]/pG$ vs stash size W when the total storage is $S = 160$. All the solid (blue) lines assume that the expected number of sampled out-of-order packets is $pR = 100$, and each line has a different value for the expected number of dropped packets $p\beta T$ as $\{20, 40, ..., 120\}$. The dotted (red) lines assume that the expected number of sampled dropped packets is $\beta pT = 100$, and vary pR as $\{20, 40...., 120\}$. The (green) lines assume that the total number of dropped and out-of-order sampled packets $p(R + \beta T) = S = 160$.

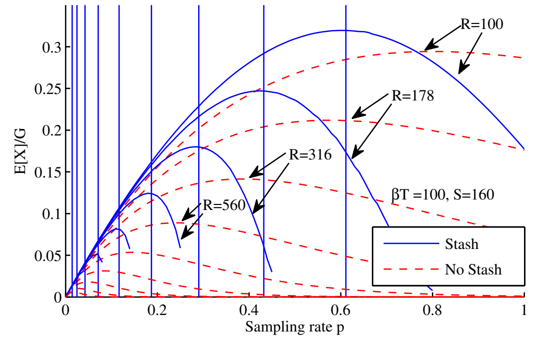


Fig. 10. Expected fraction of useful packets $E[X]/G$ vs sampling rate p when the total storage is $S = 160$ and the number of dropped packets is $\beta T = 100$. The solid (blue) lines plot equation $E[X|W = W^*]$ when the stash is of size $W^* = E[A] = pR$, and for $M = S - W^*$ and the dotted (red) lines plot $E[X|W = 0]/G$ when there is no stash. Each pair of lines has a different value for the expected number of out-of-order packets R from the logarithmicaly-spaced set $R \in \{100, 178, 316, 560, 1000, 1780, 3160, 5600, 10000\}$. The vertical lines are plotted according to (8) and represent the approximate maxima of the solid (blue) curves.

B. $E[X]$: Unconditional expected number of useful samples

We now combine the results of (2), and the distribution of L from §A to obtain the unconditional distribution of $E[X]$. Recalling that $L = A+B$ and using the Poisson approximation for B (since B is just a simple binomial distribution $\mathcal{B}(\beta T, p)$ with $p \ll \beta T$), we have

$$\begin{aligned} E[X|A] &= \sum_{b=0}^{\infty} E[X|A, b] \Pr[B = b] \\ &= \sum_{b=0}^{\infty} pG(1 - \tfrac{1}{M})^{A+b} \cdot e^{-p\beta T} \frac{(p\beta T)^b}{b!} \\ &= pGe^{-\beta pT/M}(1 - \tfrac{1}{M})^A \qquad (13) \end{aligned}$$

We can also use the Poisson approximation for A to obtain

$$\begin{aligned} E[X] &= \sum_{a=0}^{\infty} E[X|a] \Pr[A = a] \\ &= pGe^{-\beta pT/M}(\mathbf{F}(W; pR) \\ &\quad + \frac{e^{-pR/M}}{(1 - \frac{1}{M})^W}(1 - \mathbf{F}(W; (1 - \tfrac{1}{M})pR))) \qquad (14) \end{aligned}$$

where $\mathbf{F}(W; \lambda)$ is the cumulative Poisson distribution, that is $\mathbf{F}(W; \lambda) = \sum_{i=0}^{W} e^{-\lambda} \frac{\lambda^i}{i!}$. Since (14) is so complicated, we do most of our analytic work on $E[X|L = E[L]]$ in (4), and use numerical methods to work with $E[X]$ in (14).

C. Optimizing W for fixed sampling rate p

In (5) we assumed that sampling rate p and total storage S is fixed, and found the optimal sizing of stash stage W that maximizes $E[X|L = E[X]]$ (*i.e.*, the expected number of useful samples). In this section, we show qualitatively that the stash sizing in (5) obtained by working with the conditional expectation $E[X|L = E[L]]$ also applies when we work with the unconditional expectation $E[X]$ in (14). To do this, we substitute $M = S - W$ into (14) and plot the resulting $E[X]$ as a function of W in Figure 9.

From Figure 9 we make a number of qualitative observations. First, we observe that when there are fewer bad sampled packets, namely $p(R+\beta T) \leq S = 160$, the expected fraction of useful samples is maximized approximately when the stash has size W slightly larger than pR. That is, we want the stash to be slightly larger than the expected number of out-of-order packets. On the other hand, when there are many bad sampled packets *i.e.* $p(R + \beta T) \geq S = 160$, the expected fraction of useful samples is a monotonically decreasing function; it follows that maximizing the expected number of useful samples requires us to allocate all the storage to the buckets, and set the stash size to $W = 0$.

D. Optimizing p with Stash $W = pR$

Using $E[X|L = E[L]]$**.** Now, in (5) we found that when we have a stash, its optimal size is $W^* = E[A] = pR$. We find the optimal value of p at this optimal value of stash size $W = W^*$ by setting $M = S - W^*$ in (2) and (4) to obtain

$$E[X|L = E[L], W = pR] = pG\left(1 - \frac{1}{S - pR}\right)^{\beta pT} \qquad (15)$$

Now, by taking the derivative and setting it equal to zero, we find that the maxima of (15) occurs when the sampling rate is approximately

$$\begin{aligned} p^* &\approx \frac{1}{2(\rho T)^2}\left((2\rho+\beta)TS - (\rho+\beta)T \right. \\ &\qquad \left. -\sqrt{8(1-S)S(\rho T)^2 + ((2\rho+\beta)TS - (\rho+\beta)T)^2}\right) \\ &\approx \frac{S}{2\rho^2 T}\left(2\rho + \beta - \sqrt{4\rho\beta + \beta^2}\right) \end{aligned}$$

where the second approximation assumes that $S \gg 1$.

Working with $E[X]$**.** Now, we show qualitatively that the stash sizing in (5) obtained by working with the conditional expectation $E[X|L = E[L]]$ also applies when we work with the unconditional expectation $E[X]$ in (14). To do this, we

consider the two cases. For the first case, we assume that there is no stash (which we showed is optimal when $S < p(R+\beta T)$) and plot (7) as the dotted (red) lines in Figure 10. For the second case, we assume that the stash is of size $W^* = pR$, and substitute $W = W^*$ and $M = S - W^*$ into (14) and plot the resulting as the solid (blue) lines in Figure 10. We can make a number of observation from Figure 10:

• **Stash is better than no stash.** From Figure 10 we can right away observe that for a fixed value for R, βT, and S, the expected number of useful samples is higher when we use a stash of size W^* than when we have no stash (since the maxima of the solid (blue) curves are higher than the maxima of the dotted (red) curves).

• **Our approximation for p^* is good.** Furthermore, Figure 10 shows that our approximation for the optimal sampling rate p^* in (8) is quite good (since the vertical lines indeed coincide with the maxima of the solid (blue) curves). These qualitative results support our analysis using $E[X|L = E[L]]$.

Recommendations. We would like to operate the data structure at the maxima of the solid (blue) curves from Figure 10. Thus, we suggest using a sampling rate of p^* per (8), and stash of size $W^* = p^*R$.

REFERENCES

[1] Woven Systems, Inc., "EFX switch series overview," http://www.wovensystems.com/pdfs/products/Woven_EFX_Series.pdf, 2008.

[2] "NYSE shrinks time measurement to nanoseconds," http://www.computerworld.com.au/article/308952/nyse_shrinks_time_measurement_nanoseconds/.

[3] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained TCP retransmissions for datacenter communication," in *ACM SIGCOMM*, 2009.

[4] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese, "Every MicroSecond Counts: Tracking Fine-grain Latencies Using Lossy Difference Aggregator," in *ACM SIGCOMM*, 2009.

[5] "Cisco market data network overview," http://www.cisco.com/en/US/docs/solutions/Verticals/Financial_Services/md-arch-ext.html.

[6] "Cisco data center network architecture and solutions overview," http://www.cisco.com/en/US/solutions/collateral/ns340/ns517/ns224/ns377/net_brochure0900aecd802c9a4f.pdf.

[7] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM*, 2008, pp. 63–74.

[8] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving Datacenter Performance and Robustness with Multipath TCP," in *ACM SIGCOMM*, 2011.

[9] "Corvil tool minimises latency," http://www.computerworlduk.com/technology/networking/networking/news/index.cfm?newsid=5797.

[10] J. Sommers, P. Barford, N. Duffield, and A. Ron, "Accurate and efficient SLA compliance monitoring," in *ACM SIGCOMM*, 2007.

[11] J. Mahdavi, V. Paxson, A. Adams, and M. Mathis, "Creating a scalable architecture for internet measurement," in *Proc. of INET'98*, 1998.

[12] J. Sommers, P. Barford, N. Duffield, and A. Ron, "Improving accuracy in end-to-end packet loss measurement," in *ACM SIGCOMM*, 2005.

[13] S. Savage, "Sting: a TCP-based network measurement tool," in *Proc. of USENIX Symposium on Internet Technologies and Systems*, Oct. 1999.

[14] N. G. Duffield and M. Grossglauser, "Trajectory sampling for direct traffic observation," in *IEEE/ACM Transactions on Networking*, 2000.

[15] T. Zseby, S. Zander, and G. Carle, "Evaluation of building blocks for passive one-way-delay measurements," in *PAM*, 2001.

[16] T. Qiu, J. Ni, H. Wang, N. Hua, Y. R. Yang, and J. J. Xu, "Packet Doppler: Network Monitoring using Packet Shift Detection," in *ACM CoNEXT*, 2008.

[17] M. Lee, N. Duffield, and R. R. Kompella, "Not all Microseconds are Equal: Enabling Per-Flow Latency Measurements Using Reference Latency Interpolation," in *ACM SIGCOMM*, 2010.

[18] M. Bellare and D. Micciancio, "A new paradigm for collision-free hashing: incrementality at reduced cost," in *Eurocrypt97*, 1997.

[19] B. Jenkins, "Algorithm alley," Dr. Dobb's Journal, September 1997.

[20] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Transactions on Computers*, vol. 46, no. 12, Dec. 1997.

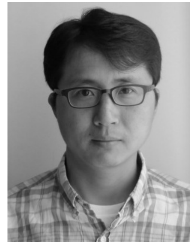
[21] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," *J. Computer and System Sciences*, vol. 58, no. 1, pp. 137–147, Feb. 1999.

[22] H. Finucane and M. Mitzenmacher, "An Improved Analysis of the Lossy Difference Aggregator," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 2, pp. 4–11, April 2010.

[23] K. Papagiannaki, S. Moon, C. Fraleigh, P. Thiran, F. Tobagi, and C. Diot, "Analaysis of measured single-hop delay from an operational backbone network," *IEEE JSAC*, vol. 21, no. 6, 2003.

[24] F. Baccelli, S. Machiraju, D. Veitch, and J. C. Bolot, "The role of pasta in network measurement," in *ACM SIGCOMM*, 2006.

[25] "Cypress Semiconductor," http://www.cypress.com/.



Myungjin Lee (M'12) is an Assistant Professor and Chancellor's Fellow in the School of Informatics at the University of Edinburgh. He received his Ph.D. degree from Purdue University in 2012, M.S. from KAIST in 2002, and B.E. from Kyungpook National University, Korea in 2000. His research focuses mainly on scalable measurement architectures and algorithms for data center networks, scheduling resources in cloud networks, and networked application monitoring and characterization.

Sharon Goldberg is an Assistant Professor in the Department of Computer Science at Boston University. Her research focuses on finding practical solutions to problems in network security. She received her Ph.D. from Princeton University in 2009, and her B.A.Sc. from the University of Toronto in 2003.

Ramana Rao Kompella (M'07, ACM M'07) is currently an Associate Professor in the Department of Computer Science at Purdue University. He directs the Systems and Networking Laboratory at Purdue that conducts research on efficient routing and transport protocols for data center networks, network performance optimizations in cloud environments and efficient router algorithmics. He received his Ph.D degree from UCSD in 2007, M.S from Stanford University in 2001, and B.Tech degree from IIT Bombay in 1999. He is a recipient of the prestigious NSF CAREER award in 2011 and has received several best paper awards in top conferences such as ACM SOCC.

George Varghese (M'99, ACM F'99) worked at DEC for several years designing DECNET protocols and products (bridge architecture, Gigaswitch) before obtaining his Ph.D. in 1992 from MIT. He worked from 1993-1999 at Washington University. He joined UCSD in 1999, where he currently is a professor of computer science. He won the ONR Young Investigator Award in 1996, and was elected to be a Fellow of the Association for Computing Machinery (ACM) in 2002. Together with colleagues, he has 16 patents awarded in the general eld of Network Algorithmics. Several of the algorithms he has helped develop have found their way into commercial systems including Linux (timing wheels), the Cisco GSR (DRR), and Microsoft Windows (IP lookups). He also helped design the lookup engine for Procket's 40 Gbps forwarding engine. He has written a book on building fast router and endnode implementations called "Network Algorithmics", which was published in December 2004 by Morgan-Kaufman. In May 2004, he co-founded NetSift Inc., where he was the President and CTO. NetSift was acquired by Cisco Systems in 2005. From Aug 2005 to Aug 2006, he worked at Cisco Systems to help equip future routers and switches to detect trafc patterns for measurement and security.