Fine-Grained Latency and Loss Measurements in the Presence of Reordering

Myungjin Lee
Purdue University

Sharon Goldberg
Boston University

Ramana Rao Kompella
Purdue University

George Varghese
UC San Diego

ABSTRACT

Modern trading and cluster applications require microsecond latencies and almost no losses in data centers. This paper introduces an algorithm called *FineComb* that can estimate fine-grain end-to-end loss and latency measurements between edge routers in these data center 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 can provide orders of magnitude better accuracy in loss and delay estimates in the presence of reordering compared to LDA.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network management

General Terms

Measurement, algorithms

Keywords

Passive measurement, latency, packet loss, reordering

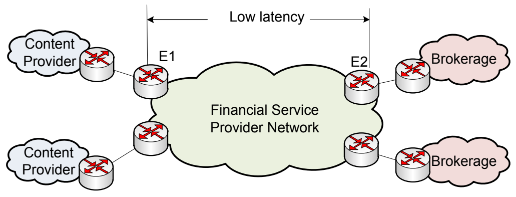
1. INTRODUCTION

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 1000's 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.*, [30]). 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* [6].

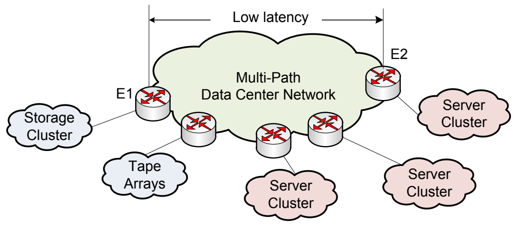
Despite the most careful selection of network components, there is no easy way for network operators to *guarantee* that congestion in switches never causes latencies to increase beyond acceptable thresholds. First, there are no traffic models for different applications that allow a manager to predict which applications can cause problems. Second, new applications must be deployed and their behavior is often unforeseen. For example, the effects of barrier-synchronized workloads overflowing switch buffers leading to packet loss and high latency was recently discovered as the well-known "in-cast" problem [29]. While solutions and work-arounds may often exist for specific problems, network operators need to perform latency measurements on a continuous basis to detect and fix such problems, either by re-routing the offending application or upgrading links to a higher capacity, or by some other means.

At a minimum, there are two types of measurements network operators typically need. First, they need *end-to-end measurements* in the network to check whether end-to-end latencies and losses are within satisfactory limits for a given customer or an application that are often specified in the form of service-level agreements (SLAs). Second, if a customer or application experiences bad performance (delay spikes or packet losses), it is important to quickly diagnose the root cause of the problem; this means obtaining *switch-level measurements* to localize the offending switch along the path. In many respects, these measurement requirements are similar to what ISPs face; the key difference, however, is that end-to-end delays in data centers are in the order of a few microseconds compared to milliseconds in ISP networks. Thus, standard approach using end-to-end active probes and tomography (for obtaining hop-level measurements) are not effective due to their huge probe requirements (*e.g.*, $\sim$10,000 per second [16]).

Recognizing these challenges, researchers have already begun to propose scalable data structures such as LDA [16] for fine-grained switch-level latency measurements. Unfortunately, detecting end-to-end latency spikes by using LDA at the switch-level is impractical, as individual switch delays are not easily summable (see §2.3). Besides, this approach incurs significant deployment cost as each switch needs to be equipped with an LDA. Therefore, scalably measuring end-to-end latency is still a requirement not satisfied with existing solutions such as LDA.



(a) Financial service provider



(b) Data center network

Figure 1: Low end-to-end latency applications.

Depending on the particular scenario, the two end points between which latency and loss measurements are required vary. For example, in a market data network architecture [2] (shown in Figure 1(a)), data feeds from content providers (*e.g.*, stock exchanges) are often provided to individual brokerages using financial service providers (FSPs). In this scenario, the FSPs may want to provide a latency SLA of a few microseconds 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, there are clusters of servers interconnected with storage servers, tape arrays and other such infrastructure [1] (as shown in 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, or even from an edge router to another edge router within a multi-rooted tree topology (*e.g.*, a fat-tree [7]).

In this paper, we consider a passive approach for measuring end-to-end delays. Instead of injecting active probes for measurement, we focus on measuring the latencies of actual packets that travel between the 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. At first glance, this problem then reduces to the same abstraction as conducting latency measurements within a switch, such as the approach taken by LDA in [16], where they measure actual packet latencies across a switch. The key difference, however, lies in the fact that LDA *crucially assumes* FIFO ordering (*i.e.*, in-order arrival) of packets between the two measurement end-points—an assumption that may not hold well in our setting.

In our end-to-end setting, we need to allow for the presence of *packet reordering* across the two measurement endpoints. Because multiple flows are present between the two, and maintaining per-flow state (for a large number of flows) is costly, our goal is to obtain aggregate measurements of all packets *across* flows. Thus, 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 by causing some packets to arrive faster than the others.

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. Anecdotal evidence suggests that many switch vendors (*e.g.*, Cisco) have internal settings by which packets can be load-balanced across multiple equal paths using packet spraying (which can reorder packets as opposed to flow hashing which preserves order). The reason these settings are never used is because standard TCP implementations are perceived to interact poorly with reordering, especially the interaction with fast retransmit and congestion control that can cause window sizes to shrink unduly by conflating loss and misordering. However, a number of researchers have been looking at creating reordering-tolerant TCP, at least for use in data centers; for example, Multipath-TCP [23] may be a point of departure for such ideas. 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 [16], which assumes FIFO packet ordering, will not work well in these environments, as it can confuse reordered packets with lost packets. 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 network edges to spot microscopic delay variations (in the order of microseconds) 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 (with synthetic workloads); our experiments indicate FineComb can achieve 10x lower relative error for latency estimates and 200x lower relative error for loss estimates compared to LDA, even under small amounts of reordering.

2. 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.

2.1 Measurement Goals

Figure 1 shows two canonical low-latency network scenarios. In both kinds of scenarios, our goal is to measure the aggregate performance between two edge routers, say $E1$ and $E2$ in Figure 1. We divide time into intervals (a few seconds) for which we are interested in obtaining performance measures. As a first step, 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. Thus our implementations need to satisfy the following constraints. We require that our data structure scale well in terms of control bandwidth, processing time, and storage. This is especially important as these metrics must be measured for each destination edge router. Of the three measures, storage may possibly be increased in a software implementation, but 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.

2.2 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 solution such as Corvil [3].

Packet filtering. Packets that arrive at a given ingress edge router will potentially exit via different destination edge routers. 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 traverse from $E1$ to $E2$ so that both $E1$ and $E2$ could precisely identify the set of packets over which the metrics need to be computed.

No header changes. Measuring latencies would be easy if we could embed a timestamp within each packet. However, IP packets do not have a 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 a large number of components in the data path of switches.

2.3 Issues with earlier solutions

Active probes: Active probes are insufficient for three reasons: First, to measure microsecond latencies, a large number of active probes need to be injected as prior work [16] indicated. Second, active probes do not measure the true delay experienced by regular data packets. Third, active probes may take one among many different paths that may potentially exist between the pairs of edge routers. See §5 for a quantitative argument.

Storing timestamps locally: An alternative is to allow the sender and receiver edge monitors to store packet digests and timestamps locally, and only to exchange these timestamps at the end of a measurement interval. However, the storage and communication overhead for this scheme is extremely high. 10 Gigabit capacity at 10% utilization between two edge routers translates to roughly 1 million packets on a per second basis, assuming an average packet of size 125 bytes. One could maintain timestamps only for a small *sample* of packets; but, as we show in §5.3, this reduces bandwidth at the cost of accuracy. While we are not certain, it appears that vendors such as Corvil [3] and NetScout [5] use variants of this approach. The lack of scalability may be indicated by the fact that the 10G Corvil solution [3] costs 90,000 U.K pounds.[1]

LDA: The LDA [16], a recent proposal for measuring fine grain delays, suggests a way of greatly increasing the number of latency samples using aggregation. LDA requires the sender to send a synchronization message at the start of every measurement interval that is injected in the same stream as the regular data packets. 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. As we show in §5.3, even a few reordered packets can cause LDA to incur a large error in both loss and latency estimation.

Why Per-Path LDA does not work ? The obvious fix to LDA is to extend it to operate on a per-path basis. Unfortunately, neither 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. Sampling a sufficiently large number of flows to ensure (with high probability) that at least one flow is sampled per path is possible. However, besides the extra multiplicative factor in memory (to cover the number of paths and the extra factor to ensure high probability coverage), the sampling scheme has a fatal flaw. 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).

[1]Of course, Corvil solution may enable more detailed analysis, but a significant cost comes from packet capture, processing and storage overheads.

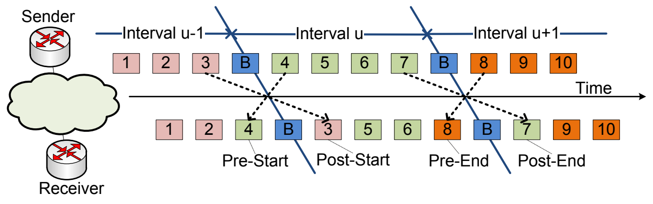


Figure 2: Four types of reordering that can occur.

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 (if no loss, *every* packet from a source to a destination is included in latency calculation).

3. FINECOMB

FineComb assumes a stream of packets going from a sender Snd (*e.g.*, $E1$ in Figure 1) to a receiver Rcv (*e.g.*, $E2$). Time is divided into measurement intervals that are marked by interval start and end messages that are transmitted from Snd to Rcv. FineComb, like LDA, starts with the following simple idea: Suppose Snd and Rcv agree on a set of packets in the stream over which they want to measure delay. Then, they could 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*). 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. But how should Snd and Rcv agree on the set of packets, in the presence of packet loss and reordering, without marking or modifying packets? This is exactly the challenge addressed by FineComb.

3.1 The challenge of reordering

In FineComb, Snd and Rcv agree upon an interval of T packets that they would like to measure delays over. 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, but we define interval as T packets for ease of exposition.) All packets 'bookended' by a pair of sync messages belong in a single interval. For convenience, we shall refer to the first sync message in an interval as an interval-start message, and the end sync message as an interval-end message.

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 end 'bookends' does not affect FineComb (or LDA), since the timestamp accumulators and counters are order-agnostic (addition is commutative). However, we must deal with 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.

Specifically, there are four types of reordering (as shown at the bottom of Figure 2) that can be problematic. 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. Similar problematic reordering could also occur around the end of the interval (analogously referred to as *post-end* and *pre-end* packets). We say $\rho = R/T$ is the *reordering rate* for the interval u, where R is the total number of reordered packets (sum of all the four types).

It is crucial to note that R is almost always much smaller than T, the number of packets sent in an interval, even if there is persistent reordering. This is because problematic reordering is confined to the reordering that occurs relative to the interval-start and interval-end messages. For example, suppose the interval-start and end packets are routed on one path (high or low latency) 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 1,000 bits, R is around 2,000 packets. By contrast, T, the number of packets sent in the interval, may be as large as 5 million.

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 the interval u.

Now, if we compare the two streams of packets that belong to an 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 above.

3.2 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*. An incremental stream digest 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, and H is a hash function. We refer to $H(pkt_t)$ as a digest. Our incremental packet digests are similar to the incremental collision-free hash functions proposed in cryptography [9]. However, since we are not operating in an adversarial setting, we can let H be a simpler hash function such as BOB [14] or H3 [25], and $\odot$ as XOR; we do not require the full power of a cryptographic hash function such as SHA-1.

The incremental stream digest has three useful properties. First, two streams containing different packets will hash to different values with high probability while packets may cancel each other out in an adversary setting. 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 incremental stream digests match at the sender and receiver. Finally, we can easily add or subtract packets from the incremental stream digest by computing the XORs of their digests with the incremental stream digest. This third property is the basis of *stash recovery* which we describe next.

2) Stash recovery: By a stash, we simply mean that we keep a copy of the timestamp, the bucket index, and incremental stream digest of a small number of packets that arrive before and after the sync messages that delimit an interval. As we have seen, R is small (say 1,000). Since these are the most likely messages to have been reordered, stash recovery simply attempts to add or subtract the incremental stream digest of each stashed message from the corresponding bucket into which that stashed message 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 the incremental stream digest for P from the incremental stream digest at the receiver. If the resulting incremental stream digest matches the incremental stream digest 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 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 cause buckets to become 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 over

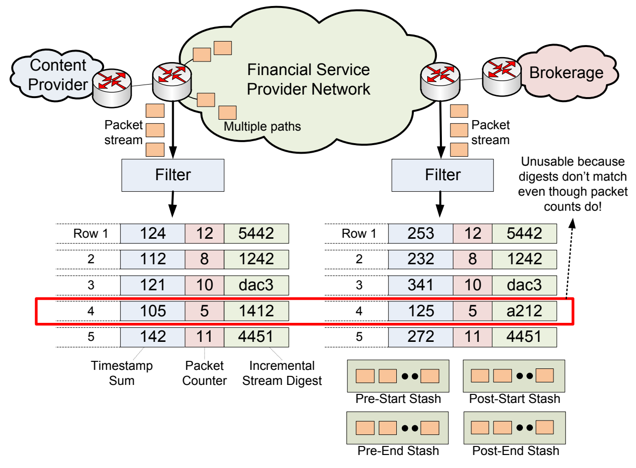


Figure 3: Example of FineComb. The four stashes cater to the four types of reordered packets.

which measurements are computed is a key question that our later analysis will address.

3.3 Basic FineComb without a stash

We start by describing FineComb without a stash. Basic FineComb (as shown in Figure 3) uses M buckets, each containing a timestamp accumulator, counter, and incremental stream digest. 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 $TS[i]$ represent the timestamp accumulator, $C[i]$ the packet counter, and $D[i]$ the incremental stream digest for ith bucket and M represent the total number of buckets.

```
1: procedure UPDATE STATE($pkt$, $\tau$)
2:     $D \leftarrow compute\_hash(pkt)$          → Digest
3:     $i \leftarrow D \ mod \ M$
4:     $TS[i] \leftarrow TS[i] + \tau, C[i] \leftarrow C[i] + 1$
5:     $D[i] \leftarrow D[i] \odot D$          → $\odot$ could be XOR
6: end procedure
```

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 incremental stream digests at 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:     end if
6: end for
7: $Average \ delay = D/N$
```

The main difference compared to LDA's delay estimation algorithm is the requirement of an extra check for a match of the sender and receiver packet digests; just matching the packet counters alone is not sufficient.

2) Estimating standard deviation: We compute standard deviation in a similar fashion using a technique introduced in [8]. Conceptually, we could maintain an additional counter to which each sampled packet's timestamp is added or subtracted with equal probability $1/2$. Note that we need both the sender and receiver to agree on the same decision (of adding or subtracting) consistently, that can easily be achieved if the decision is based on the packet hash itself (*e.g.*, 1 or 0 in first bit position could indicate addition or subtraction). Subtracting the sender and receiver counter and then squaring leads to an unbiased estimator for delay variance [8]. Rather than wasting memory with an extra counter per bucket to measure variance, we use a trick used in LDA where existing delay buckets are paired and subtracted to simulate the adding or subtracting with equal probability.

3) Loss measurement: Loss measurement becomes difficult in the presence of reordering. Whereas the LDA operated in a setting where there was no reordering, so that a single counter at the sender and receiver suffices, FineComb must try to disentangle reordering from real loss. To see why this is hard, consider what happens at the end of an interval for a particular bucket if the sender-side counter is smaller than the receiver-side counter. When there is no reordering (as in the scenarios the LDA was designed for), this is impossible. However, it can easily happen if a few packets drift from one interval to the previous interval (*i.e.* pre-start packets that overtake the interval-start message). These packets are not lost: they are simply accounted for in the bucket of the previous interval.

We use stash recovery (detailed description next) to "clean up" the effects of reordering wherever possible. If all effects of reordering are removed, 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:     end if
6: end for
7: $loss \ rate = L/N$
```

Note that the algorithm still checks whether a sender counter is greater than the corresponding receiver counter. This is because stash recovery can be imperfect. Further, if a lost packet and 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 messages $P1$ and $P2$ that were hashed into bucket i that were not counted in this interval. Suppose further that a third packet $P3$ that hashes into bucket i is lost. 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 bucket i is not just useless from the point of view of calculating delay, the algorithm 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 are trying to measure small losses, this is potentially serious. However, with careful sizing of the sampling probability (as we show later in §5) the probability of both a lost, and a reordered packet hashing to the same bucket is even smaller.

3.4 Managing the stash

We now describe the details of adding and recovering a stash. Recall that the stash stores individual timestamps and digests for the packets that are most likely to be problematically reordered. We assume that only the receiver keeps a stash, that consists of W *entries*. 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. The stash 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 timestamps 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. Similarly, 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 stops populating when it is full. The other two stashes are managed similarly, except they wait for interval-end instead of interval-start.

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 incremental stream digest. If the sender's and receiver's incremental stream digest 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 (if the packet is from the pre-start stash or post-end stash) 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:      $D_r \leftarrow D_r[i], TS_r \leftarrow TS_r[i], C_r \leftarrow C_r[i]$
4:      for all $(D, \tau, k) \in \mathbb{Z}$ do
5:          if $k =$ pre-start or $k =$ post-end then
6:              $D_r \leftarrow D_r \odot D, TS_r \leftarrow TS_r + \tau, C_r \leftarrow C_r + 1$
7:          else
8:              $D_r \leftarrow D_r \odot D, TS_r \leftarrow TS_r - \tau, C_r \leftarrow C_r - 1$
9:          end if
10:     end for
11:     if $D_s[i] = D_r$ then
12:         $D_r[i] \leftarrow D_r, TS_r[i] \leftarrow TS_r, C_r[i] \leftarrow C_r$, return
13:     end if
14: end for
```

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).

3.5 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 §4) allow us to configure the sampling rate optimally. In practice, however, we do not know these values *a priori* and may change over time. LDA also faces a similar problem with loss rate not being known, 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 $(\beta_{max}, \rho_{max})$. 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 rates may produce no valid delay or loss estimates.

In FineComb, we use four banks, each using 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 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 maximum count among counts from four buckets in the same row (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 us with 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. Intuitively, this heuristic uses the observation that rate estimates are typically most accurate when they are closest to what the bank is tuned for.

4. SETTING PARAMETERS

In the following analysis, our goal is to choose a sampling rate p, and stash size W that will maximize 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 have formally proved the results in this section, for brevity, we only state the main theorems, results, and proof sketches. Additional proofs appear in [19].

4.1 Expected number of useful samples

Since our goal is to maximize $E[X]$, the expected number of useful samples we can extract from FineComb, 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 in with the boundaries of interval u (see §3.1 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 $G \leq (1-\beta)T$ with equality when $R = 0$, so that there are no packets that are problematically reordered. Packets can become bad due to loss, or problematic

[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 receive requires $M + W$ storage.

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 stash recovery. We can prove that the expected number of useful samples is

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

where, following [13], 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 [13], they show that K is distributed as

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

where $S(L, k)$ is a Stirling number of the Second Kind. Using (3), we obtain

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

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

$$E[L] = \beta pT + \max\{0, pR - W\} \tag{4}$$

Working with the conditional expectation. Because the distribution of L is quite complicated, 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 $E[X]$.

4.2 Optimizing stash W for fixed sampling p

First, we would like to optimize the ratio between the LDA size and the stash size to maximize the expected number of useful samples $E[X]$, using the fact that $S = M + W$ where S is fixed and sampling rate p is fixed. 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 $E[X|L = E[L]]$:

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

This holds even when we work with $E[X]$ (rather than just $E[X|L = E[L]]$). Details can be found in [19].

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.

4.3 Optimizing sampling rate p.

No stash. Per (5) we now consider the case where we have no stash (*i.e.*, $W = 0$). We can show that the optimal sampling rate is

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

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

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

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

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

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 [19], 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 that the expected number of samples 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 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 avoided it.

5. 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 same total memory for a 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.

5.1 Evaluation methodology

We built a custom simulator in C++ for evaluating a prototype of our measurement solution. 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, measurement interval. Our simulation environment is deliberately kept similar to the one used by the authors in [16] 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 exists no such publicly available data center latency traces. Prior work [16] used the Weibull delay distribution model empirically verified to mimic the distribution of delays within a backbone router by Papagiannaki *et al.* in [22]. While we use mainly Weibull distribution (and Pareto for diversity) 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 multibank scenario (refer to §5.4 for more detail). The delay for each packet is drawn from a Weibull distribution, which has cumulative distribution function $P(X \leq x) = 1 - e^{(-x/\alpha)^{\beta}}$ with α and β representing the shape and scale of the graph respectively. We use [22]'s recommended shape parameter $0.6 \leq \alpha \leq 0.8$ in all our simulations (mostly, we used $\alpha = 0.6$). 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 about $10\mu s$. (Path latencies in data centers may range from 10–100 μs, so our setting simulates close to the finest granularity.) We show our results in the form of relative error, so exact delay average does not matter. For delay distribution, we use Weibull (and Pareto) with shape parameter 0.6 and scale adjusted to obtain mean delay of $10\mu s$. We simulate 5,000,000 packets, with an average packet size of 250 bytes (similar to [16]), over a 10 Gbps bottleneck capacity with an inter-arrival time of 0.2μs—transmission time for 250 bytes at 10Gbps is $0.2\mu s$. All our simulation results are average across 10 runs.

Reordering model. An important parameter in our simulation 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 process similar to that of packet loss; in fact, there exists no generative model that we are aware of that we can use in our simulation. We note once again that reordering within the interval does not affect either LDA or 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 §3.1. 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 §4 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 Equation 8 is computed assuming all these different individual reordering components are the same. To make our provisioning strategy consistent with 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 $10^{-3}T$ (T being total number of packets); equivalently, the reordering rate ρ varies from $4 \cdot 10^{-6}$ to $4 \cdot 10^{-3}$, translating to roughly 50 to 5,000 packets before the interval-end message. 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.

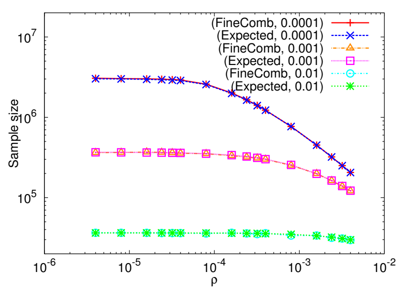
Resource configuration. We allocate a total of 1,000 buckets for FineComb. To simulate cases with and without stash, we assume stash elements are of the same size as bank elements (for simplicity). We use 64 bits from a 160-bit SHA-1 hash function for packet digests. To make things fair, we equalize 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 incremental stream digest). Furthermore, while FineComb is asymmetric (only the receiver maintains stashes), the LDA is symmetric. Thus, memory is allocated as follows: LDA gets $1.5(M + W/2)$ buckets at sender and receiver, where M is number of FineComb buckets and W is stash size.

5.2 Assessing FineComb

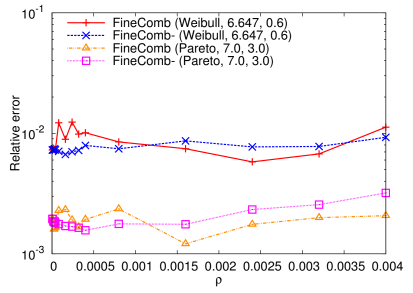
Expected number of samples. In our first experiment, we wish to understand how tight the theoretical bound on the number of useful samples is, at the optimal sampling probability. In Figure 4(a), we plot the expected number of samples according to the analytical bound given in Equation 2 (curve titled '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 (0.0001, 0.001, 0.01). Clearly, as we increase the loss rate from 0.00001 to 0.001, the number of effective samples over which the delay estimates are computed reduces all the way from almost 3 million packets at loss rate 0.0001 (0.01%), to about 40,000 packets at 0.01 (1%) loss rate. 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 loss causes more FineComb buckets to become useless, causing the expected number of samples to decrease.

In all cases, we observe that analytically expected number of samples matches quite well with what we found empirically (the curves are virtually indistinguishable); the difference between expected and empirical is of the order of a few hundreds, with the predicted number of samples slightly smaller than what we found empirically.

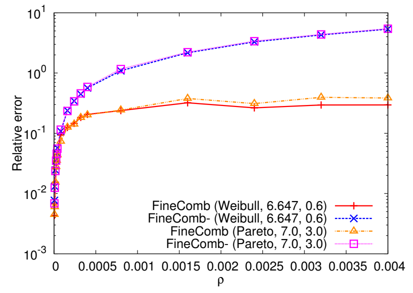
Latency estimates. Next, we show the average relative error of mean delay and loss estimates, as we vary the reordering rate ρ in Figure 4. We show the results comparing FineComb and FineComb- (FineComb without the stash) for two different distributions, Weibull



(a) Expected number of samples



(b) Mean delay estimation, β = 0.01



(c) Loss rate estimation, β = 0.0001

Figure 4: Expected number of samples obtained by FineComb, and relative error of mean delay and loss estimates in the presence of forward reordering under different distributions. We show both FineComb and FineComb- for comparison.

and Pareto with shape and scale parameters adjusted to ensure similar mean latency of $10\mu s$. 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% for either of the two distributions, 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 30% for Pareto distribution and 9% Weibull distribution across all ρ values.

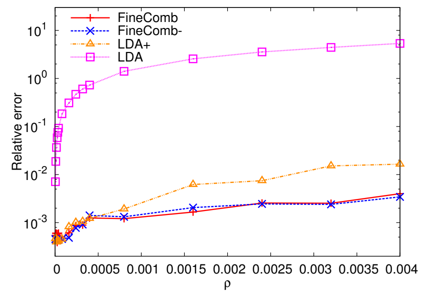
As predicted by our analytical work in [19], FineComb provides about 15-30% more useful samples than FineComb- (that has no stash). While more samples should lead to better delay estimates, the improvement in the delay estimate depends heavily on the specific delay distribution; that is, some distributions require fewer samples to obtain accurate estimates (*e.g.*, to take things to an extreme, a uniform distribution requires only a small number of samples for excellent accuracy in delay estimates).

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. Note that the delay distribution itself does not effect loss rate estimation (the little difference visible is caused by different random number seeds).

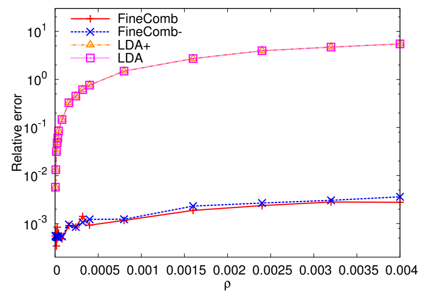
5.3 Comparison with other solutions

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 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 from the sender 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.



(a) Forward reordering



(b) Backward reordering

Figure 5: Average relative error of mean delay estimates comparing FineComb with LDA. β = 0.0001.

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- with no stash (note from §4 that adding the stash only *increases* the number of good samples produced by the FineComb). Combining (7) and (8), we find that when FineComb has S buckets and no stash, it produces

$$E[X_{FineComb}] = S\frac{G}{\beta T + R}e^{-1} \qquad (9)$$

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 $S_{sample} = S$, and 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 in the interval, 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.

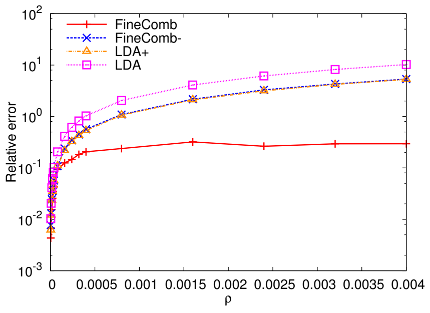
3) LDA for latency estimates: In Figure 5, we plot the relative error of mean delay estimates for four solutions, namely LDA, LDA+ (a small refinement of LDA we discuss later), FineComb and FineComb- for different reordering rates and reordering models. For this set of experiments, we choose optimal stash size configurations and sampling probabilities (for LDA, as recommended in [16]) for all solutions.

The main observation from the graphs is that, beyond small levels of reordering, LDA consistently performs the 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 to 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.*, 1 second).

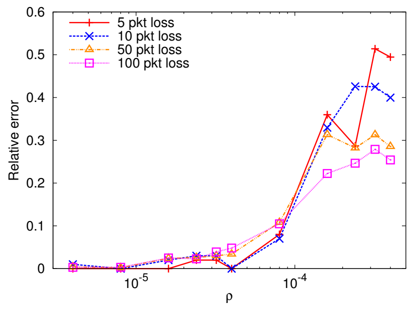
LDA+ is a simple refinement of LDA which effectively ignores the set of buckets where the sender's timestamp sum is higher than the receiver timestamp sum (which could be caused by a situation like the one we described above) and results in *a negative delay* contributed by that bucket. This clearly helps solve most of the problems in the forward reordering case (where extra packets drift into the interval), as reflected in the better relative error for LDA+ in Figure 5(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 backward reordering case, since in this case, we cannot easily detect (using a simple elimination scheme as before) and eliminate buckets that are anomalous because of reordering. Thus for the lower set of graphs, we can see that LDA+ has the same accuracy level as the LDA.

In all cases, we can observe that both FineComb and FineComb- perform consistently better than LDA even under high loss and reordering rates. We can observe that 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 why LDA's mean delay estimates are quite bad 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 6(a), we plot the relative error in estimating loss rate (for $\beta = 0.0001$). As we can see from the figure, 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) Low loss, β = 0.0001



(b) Microscopic loss, 5-100 lost packets

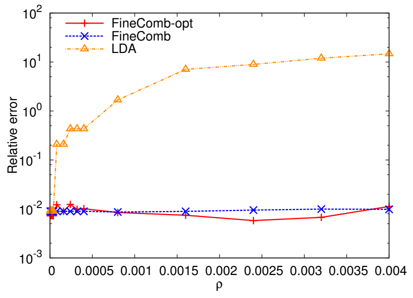
Figure 6: Relative error of FineComb at detecting low to microscopic loss rates.

Microscopic losses. While 10-30% error in estimating loss rates as low as 0.0001 is good, our goal was to also be able 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.*, say 500 packets, *i.e.*, $\rho = 10^{-4}$) is possible only with extremely high rates of sampling (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 1 packet loss in 1 million (our definition of microscopic losses). In Figure 6(b), we see that, even though the relative error of FineComb's loss estimates becomes progressively worse as reordering increase, the estimates are well within 10% for reordering rates up to 10^{-4} (500 reordered packets), *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.

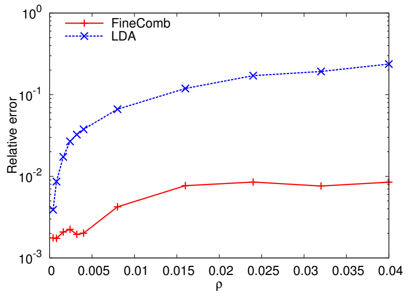
5.4 Handling unknown loss and reordering rates

We have already demonstrated 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.

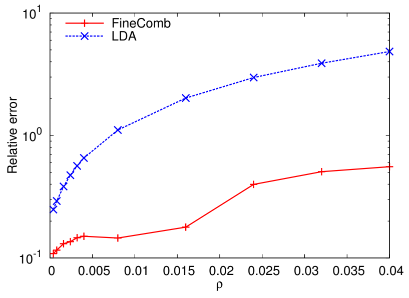
We use Weibull delay distribution model as well as a real trace to compare the efficacy of 4-bank FineComb with a two-bank LDA under unknown loss and reordering rates. First, for Weibull delay distribution model, average latency is set to $10\mu s$ and β is set to 0.01. Second, the trace collected from an ingress and an egress interface of a router by the authors in [27] contains about 2.4 million packets over about two and half minute interval which experienced queueing delay, packet loss, and so on. Since OC-3 link



(a) Mean delay estimation with β = 0.01 using Weibull delay distribution



(b) Mean delay estimation using a real trace



(c) Loss rate estimation using a real trace

Figure 7: 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 forward reordering scenario.

(155 Mbps) was used when the trace was collected, we do not use a second measurement interval which only has a few thousands of packets. Instead, we group the packets into measurement intervals each of which consists of 0.2 million packets. The average loss rate of the trace is about 0.24%, but each measurement interval 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. Thus, we adjust packet reordering rate from 0.0004 to 0.04 for the evaluation.

For 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 0.01.

Figure 7(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 given the knowledge of loss and reordering rates. The results for other loss rates, and for the backward reordering case, are quite similar to the curve of FineComb (and hence, omitted). From the figure, we can observe that LDA performs worse than FineComb (as we have observed before) even in the case of multiple banks. At extremely low reordering rates, the estimates of LDA are quite accurate, but they 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 results are reasonably accurate with a relative error of less than 1% under almost all conditions. Standard deviation estimation also shows similar trend with mean estimation, so we omit the exact graph for brevity. As a summary, while LDA achieves better accuracy than FineComb when ρ is extremely small, FineComb has at least two orders of magnitude less errors than LDA from $\rho = 0.0008$, and about 11-13% relative errors are obtained by FineComb across all reordering rates.

In Figure 7(b), we observe the similar pattern shown in Figure 7(a). However, compared to the results of Figure 7(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. Due to the high latencies, 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 from around 0.0024 reordering rate. Again, FineComb achieves a relative error of less than 1% under all conditions. Similarly, for standard deviation estimation, LDA shows an order of magnitude higher relative error than FineComb for most reordering rates.

In Figure 7(c), we show the relative error of the loss rate estimation. FineComb's relative error shows less than 20% up to 0.016 reordering rate. After the reordering rate, the relative error of FineComb is comparatively worse at around 55%, but LDA is completely unusable across almost all the rates.

6. IMPLEMENTATION

With 1000 buckets and 1000 stash entries, FineComb should take a small percentage of a low end 10mm×10mm networking ASIC using a 400-MHz 65nm process. Key to a small footprint is a cheap version of an incremental stream digest using a loop-unrolled Rabin hash. A quicker path to deployment *today*, however, is using high-end FPGAs such as the NetFPGA [4]. For time synchronization, the boards need to have GPS chipsets (fairly cheap today), the solution used by monitors such as Corvil [3].

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 $O(M2^{W/M})$, where W is the size of the stash and M is the number of buckets. 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, the number of buckets 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 Intel processor 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 will need to be done on board, functions such as stash recovery can be implemented in software on the PC. Implementing FineComb on boards (based on either FPGAs or network processors) 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 [3]. 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.

7. RELATED WORK

While network latency measurements is a rich area of research in the Internet with several tools proposed in the past to obtain latency measurements, the fundamental focus on fine-grain microscopic latency and loss measurements, makes most of these tools not suitable for the task at hand. Scalable performance measurements for data center environments is a relatively less studied field.

The standard approach for conducting latency measurements in the wide area is to inject active probes (*e.g.*, using ping and other tools such as [28, 21, 27, 26]) and calculate the round-trip time of the packet. We have discussed the problems with active probes in §2.3. Router-based passive measurements is yet another active area of research [11, 12, 31, 24, 15]. They focus mainly on flow measurements such as number of packets and bytes, and not on latency and loss estimation. In [20], the authors propose a measurement-friendly network architecture; the goal is to infer router characteristics with the help of end-to-end measurements. Our goal is to measure end-to-end characteristics with support at the end points, however. There are a few prior efforts (*e.g.*, [10, 32]) where researchers proposed simple router extensions for latency measurements that are somewhat similar to the local timestamps idea discussed in §2.3, and hence share similar problems.

Perhaps the most relevant research effort to ours is a recent data structure called LDA proposed by Kompella *et al.* in [16], and an incremental deployment architecture in [17]. Given the close similarity, we discussed it at length in the paper, and compared the performance of FineComb with LDA. In [18], Lee *et al.* describe a per-flow switch-level latency measurement architecture. In our work, we focus on measurements across flows, so our goals are different from theirs.

8. CONCLUSIONS

Measurement tools are badly needed to determine fine-grain latencies and losses that can affect application SLAs in data center environments. Existing scalable approaches such as LDA designed for switch-level measurements works poorly for end-to-end measurements in the presence of packet reordering which actually happens in IP networks. We describe a simple yet scalable data structure called FineComb 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 in order to measure loss precisely to a few parts in a million. 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.

Acknowledgments

The authors thank Kirill Levchenko, Michael Mitzenmacher, Zvika Brakerski, and the anonymous reviewers 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.

9. REFERENCES

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

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

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

[4] NetFPGA. `http://www.netfpga.org`.

[5] NetScout. `http://www.netscout.com`.

[6] NYSE shrinks time measurement to nanoseconds. `http://www.computerworld.com.au/article/308952/nyse_shrinks_time_measurement_nanoseconds/`.

[7] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *ACM SIGCOMM* (2008), pp. 63–74.

[8] ALON, N., MATIAS, Y., AND SZEGEDY, M. The space complexity of approximating the frequency moments. *J. Computer and System Sciences 58*, 1 (Feb. 1999), 137–147.

[9] BELLARE, M., AND MICCIANCIO, D. A new paradigm for collision-free hashing: incrementality at reduced cost. In *Eurocrypt97* (1997).

[10] DUFFIELD, N. G., AND GROSSGLAUSER, M. Trajectory sampling for direct traffic observation. In *IEEE/ACM Transactions on Networking* (2000).

[11] ESTAN, C., KEYS, K., MOORE, D., AND VARGHESE, G. Building a Better NetFlow. In *ACM SIGCOMM* (2004), pp. 245–256.

[12] ESTAN, C., AND VARGHESE, G. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems 21* (2003), 270–313.

[13] FINUCANE, H., AND MITZENMACHER, M. An analysis of the lossy difference aggregator. *unpublished* (2009).

[14] JENKINS, B. Algorithm alley. Dr. Dobb's Journal, September 1997.

[15] KOMPELLA, R. R., AND ESTAN, C. The Power of Slicing in Internet Flow Measurement. In *ACM/USENIX IMC* (May 2005).

[16] KOMPELLA, R. R., LEVCHENKO, K., SNOEREN, A. C., AND VARGHESE, G. Every MicroSecond Counts: Tracking Fine-grain Latencies Using Lossy Difference Aggregator. In *ACM SIGCOMM* (2009).

[17] KOMPELLA, R. R., SNOEREN, A. C., AND VARGHESE, G. mPlane: An architecture for scalable fault localization. In *ACM ReARCH* (2009).

[18] LEE, M., DUFFIELD, N., AND KOMPELLA, R. R. Not all Microseconds are Equal: Enabling Per-Flow Latency Measurements Using Reference Latency Interpolation. In *ACM SIGCOMM* (2010).

[19] LEE, M., GOLDBERG, S., KOMPELLA, R. R., AND VARGHESE, G. Fine Comb: Measuring Microscopic Latencies and Losses in the Presence of Reordering. Techincal report CSD TR 10-009, Purdue University, 2010.

[20] MACHIRAJU, S., AND VEITCH, D. A measurement-friendly network (MFN) architecture. In *Proceedings of ACM SIGCOMM Workshop on Internet Network Management* (Sept. 2006).

[21] MAHDAVI, J., PAXSON, V., ADAMS, A., AND MATHIS, M. Creating a scalable architecture for internet measurement. In *Proc. of INET'98* (1998).

[22] PAPAGIANNAKI, K., MOON, S., FRALEIGH, C., THIRAN, P., TOBAGI, F., AND DIOT, C. Analaysis of measured single-hop delay from an operational backbone network. *IEEE JSAC 21*, 6 (2003).

[23] RAICIU, C., PLUNTKE, C., BARRE, S., GREENHALGH, A., WISCHIK, D., AND HANDLEY, M. Data center networking with multipath TCP. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Networks* (2010).

[24] RAMACHANDRAN, A., SEETHARAMAN, S., FEAMSTER, N., AND VAZIRANI, V. V. Fast monitoring of traffic subpopulations. In *ACM/USENIX IMC* (2008), pp. 257–270.

[25] RAMAKRISHNA, M., FU, E., AND BAHCEKAPILI, E. Efficient hardware hashing functions for high performance computers. *IEEE Transactions on Computers 46*, 12 (Dec. 1997).

[26] SAVAGE, S. Sting: a TCP-based network measurement tool. In *Proceedings of USENIX Symposium on Internet Technologies and Systems* (Oct. 1999).

[27] SOMMERS, J., BARFORD, P., DUFFIELD, N., AND RON, A. Improving accuracy in end-to-end packet loss measurement. In *ACM SIGCOMM* (2005).

[28] SOMMERS, J., BARFORD, P., DUFFIELD, N., AND RON, A. Accurate and efficient SLA compliance monitoring. In *ACM SIGCOMM* (2007).

[29] VASUDEVAN, V., PHANISHAYEE, A., SHAH, H., KREVAT, E., ANDERSEN, D. G., GANGER, G. R., GIBSON, G. A., AND MUELLER, B. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *ACM SIGCOMM* (2009).

[30] WOVEN SYSTEMS, INC. EFX switch series overview. `http://www.wovensystems.com/pdfs/products/Woven_EFX_Series.pdf`, 2008.

[31] YUAN, L., CHUAH, C.-N., AND MOHAPATRA, P. ProgME: towards programmable network measurement. In *ACM SIGCOMM* (2007).

[32] ZSEBY, T., ZANDER, S., AND CARLE, G. Evaluation of building blocks for passive one-way-delay measurements. In *PAM* (2001).