2010

FineComb: Measuring Microscopic Latencies and Losses in the Presence of Reordering

Myungjin Lee
Purdue University, mjlee@purdue.edu

Sharon Goldberg
Boston University

Ramana Rao Kompella
Purdue University, kompella@cs.purdue.edu

George Varghese
UC San Diego

Report Number:
10-009

FineComb: Measuring Microscopic Latencies and Losses 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 loss and latency at monitoring devices between edges of a data center. 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 LDA that need to be deployed in hardware at every router and link, FineComb can be deployed today using commodity boards. The challenge for FineComb is to deal with persistent reordering that can occur in a data center network using equal cost path splitting. Without 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 easily be implemented in software or hardware, and provides significantly more accurate loss and delay estimates in the presence of reordering than LDA.

1. INTRODUCTION

Recent trends in data centers have led to requirements for *microsecond* latencies. While multimedia applications (*e.g.*, gaming, IPTV, VoIP) are content with latencies under 200 msec, modern financial trading and various cluster applications can benefit from latencies under 100 μsecs. 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. It is possible to deploy switches (*e.g.*, [31]) with 10 μsecs latency. Wall Street traders also vie with each other to get the best links from the Dow Jones stock feed to their data centers in Manhattan [22].

Despite the most careful attention paid to choosing network components, there is no way to ensure that congestion in switches does not increase latencies beyond acceptable thresholds. First, there are no traffic models for data center applications that allow a manager to predict which applications can cause problems. Second, new applications must be deployed and their behavior is often unforeseen. Third, a number of data centers such as Microsoft and Google wish to use commodity switches with small buffers. Fourth, applications are typically handled by the IT group which often coordinates poorly with the network infrastructure group.

Thus, it is extremely common for data centers to experience transient or periodic performance violations (*e.g.*, increased latency, dropped packets) which can exceed application SLAs. For example, many data centers experience what is termed a microburst [5]; intuitively, this refers to a short burst of traffic that causes a switch to exceed its buffers. This is often caused by a single uncontrolled application that simply needs to be controlled or relocated. A second example is the in-cast problem, where a single request causes multiple responses from several servers that are synchronized to cause a similar burst at a switch. While solutions have been proposed to the in-cast problem [30], it is still important to have measurement methodology to rapidly diagnose these problems.

The setting. We focus on tools to measure fine-grain latency and detect dropped packets in the following setting. We assume a data center network (managed by a separate group) that provides network infrastructure service to an IT group in charge of applications. We wish to deploy tools for the network group that can be used at taps (or routers) at the edges of the network to monitor traffic between edge points. For example, in Figure 1, we assume we have a monitor (tap and software) that taps all traffic entering E1, and a similar one for traffic leaving E2. Internally, there could be several core routers (*e.g.*, C1 and C2) along the path between these edge routers.

Fundamental to our problem definition is the assumption of significant multipathing (and hence, *persistent reordering*) within the network. For example, in Figure 1, we show two such paths between the edge routers E1 and E2. Multipathing using ECMP is very common in data center networks [8] because of the need to provide more bisection

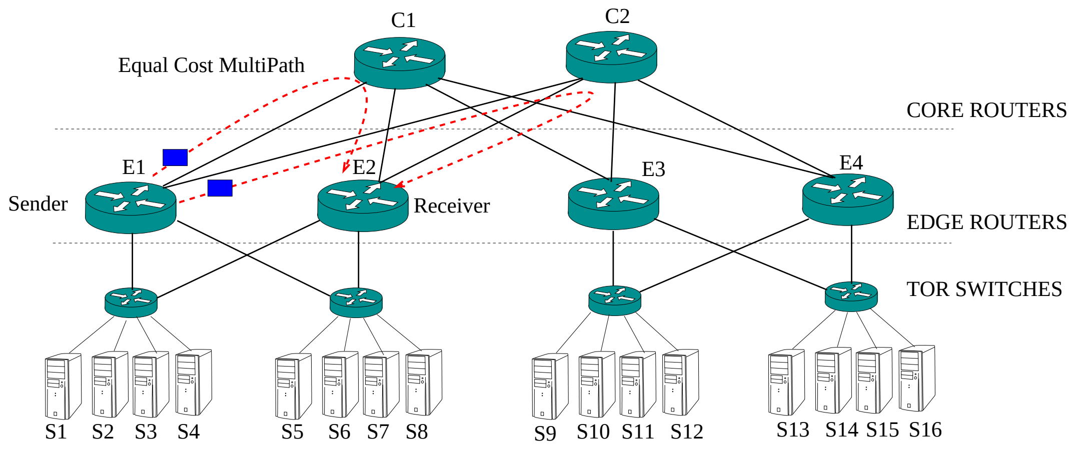


Figure 1: Typical data center architecture with edge and core routers. Two paths possible between edge routers E1 and E2 are typically load-balanced by hashing the source and destination IP address combinations.

bandwidth, and because of the use of redundant levels in common multi-rooted tree topologies.

For example, traffic from server $S1$ to $S5$ can be routed through the link to C1, while traffic from $S2$ to $S6$ can be routed through the link to C2 because ECMP hashes based on IP source-destination pairs. Now, ECMP does not reorder packets within TCP flows, so, the tap could keep latency measurement state on a per TCP flow basis. However, keeping track of the latency for potentially 100,000 simultaneous connections could require too much state in the monitor, especially in a hardware implementation. Thus it is more efficient for each edge router such as E1 (or a corresponding tap) to keep aggregate latency state for each egress edge router such as E2. However, such aggregation inherently introduces the possibility of reordering as it mixes in traffic flowing over potentially two different paths. Further, this reordering can be persistent.

To get an idea of how much reordering there can be, suppose the delay from $S1$ to $S5$ via C1 is 10 μsecs and the delay from $S2$ to $S6$ is as bad as 100 μsecs. If the link speeds are 10 Gbit and the average packet size is 1,000 bits (125 bytes), the amount of reordering can be as high as (100 - 10) $\times$ 10 = 900 packets. In other words, for reasonable parameter settings, a single packet sent from E1 to E2 can be overtaken by (or overtake) 900 other packets. Note that while the paths chosen by ECMP are equal in cost, the cost measure used in ECMP does not reflect transient congestion which can make two equal-cost paths differ widely in latency.

In this setting, imagine that application traffic from $S1$ to $S5$ has poor end-to-end latency. The IT department immediately blames the network for poor performance. It can greatly help the network managers to have a simple tool that determines that all network delays from E1 to E2 have a very low average delay and small standard deviation, and almost no dropped packets. This suggests that the likely cause of poor performance is at the hosts (*e.g.*, because of bad VM performance). If, however, the managers find large delays or large standard deviations (for example, if one path is significantly worse) then the manager can use more labor-intensive means to narrow down the problem to a path, and ultimately to an offending component.

By contrast today, managers can attempt to ascertain the health of their data centers only by sending active probes between edges. However, it can be shown that active probes require inordinate amounts of probe bandwidth to achieve microsecond accuracy. A recent paper [18] proposes a simple algorithm called LDA that is more effective than active probing. However, LDA requires FIFO delivery of messages which is clearly inapplicable in our edge-to-edge setting where reordering is a fact of life. Further, [18] proposes deploying LDA at every router, which is hard.

Our paper describes an efficient algorithm that can be implemented at the edges of a data center network to spot microscopic delay violations (in the order of microseconds) and losses (10s per million) with small amounts of state and processing costs.

2. PRELIMINARIES

We describe why reordering must be considered, the basic requirements of a measurement scheme, and a set of existing solutions that do not work well.

2.1 Reordering

Figure 1 shows a canonical data center architecture. While different designs have been proposed in the academic literature (*e.g.*, [15, 8]), most commercial data centers use a similar model. Such multi-rooted tree topologies typically provide multiple paths between every pair of servers. Traffic engineering policies in data centers often rely on simple load balancing techniques such as equal-cost multipathing (ECMP) that are natively provided by commodity routers.

Even though ECMP does not load balance traffic within a single TCP flow, the need to keep aggregate statistics at the edge-to-edge level will result in reordering of the edge-to-edge flow. Recent load balancing proposals can further exacerbate reordering. For example, [9] proposes dynamically moving large TCP flows between equal-cost paths in response to congestion. Flowlet switching [27] even proposes mechanisms to allow load balancing within a single TCP flow. Thus, any edge-to-edge measurement scheme must be resilient to persistent misordering within the network.

2.2 Measurement requirements

Assume a manager wishes to measure the performance between two edge routers, say $E1$ and $E2$ in Figure 1. We divide time into intervals (1 msec to 1 second) for which we are interested in obtaining performance measures. Besides average latency, the mechanism should measure the variance of latency and the loss rate between these edge routers. For most of this paper, we assume hardware implementations to keep up with high line rates; however, we briefly discuss software implementations. Thus we seek implementations that are scalable in terms of control bandwidth, processing time, and storage. This is especially important as these metrics for each source edge router must be multiplied by the number of destination edge routers. Of the three measures, storage may possibly be increased in a software implementation.

We assume some simple way to determine which packets are destined to or from a particular edge router, for example by prefix matching. 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 [2].

There could potentially exist many different paths between edge routers $E1$ and $E2$ as we discussed before. A natural question to ask is whether it is not possible to provide performance measures for each path. We suggest this is problematic for the following reasons. In a multilevel topology, load balancing can occur at many points, in particular at routers downstream from the measurement point. Overloading the edge router to keep track of these multiple downstream paths and the precise way the load balancing is achieved (*e.g.,* the hash function) is overly complex for both the mechanism and managers. Further, we posit that network operators are often concerned with one main latency measure across different edge routers; the subtleties about which paths are experiencing how much latency is often a second-order concern that arises once a problem is detected. Thus, in this paper, we focus on measuring the average latency across *all* paths, if multiple paths exist between a pair of edge routers. We reiterate that a high standard deviation is a signal that some path(s) has a higher than average latency.

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. Second, active probes do not measure the true delay experienced by regular data packets. Third, active probes may take one among many different ECMP paths that may potentially exist between the pairs of edge routers. We provide a more quantitative argument in the evaluation.

Embedding packet timestamps within packets: Measuring delay 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 end-to-end delays. Adding a new field is unlikely to happen. Second, even if switch and router vendors can be persuaded to put timestamps within packets, packet timestamps cause significant overhead, especially for minimum size packets.

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 bottleneck capacity between two edge routers translates to roughly 10 million packets on a per second basis, assuming an average packet of size 125 bytes. Even assuming 10% utilization, exchanging a million timestamps per second is expensive. One could maintain timestamps only for a small *sample* of packets; this reduces bandwidth but at the cost of accuracy of latency estimates as we show briefly in the evaluation. Intuitively, the standard error of the delay estimator decreases with the square root of the number of samples; reducing the standard error to microseconds requires a large number of samples. While we are not certain, it appears that vendors such as Corvil [2] and NetScout [4] use variants of this approach. The lack of scalability may be indicated by the fact that the 10G Corvil solution [2] costs 90,000 U.K pounds.

LDA: A recent proposal called LDA [18] 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 receive compute delay estimates over the same set of packets. The FIFO assumption is reasonable if LDA is used to measure average latencies across interface pairs *within* routers. While routers routinely use load-balancing internally, packets are resequenced before transmission at the output interface.

Unfortunately, in our setting, the FIFO assumption does not hold because load balancing is done *across* routers. A few reordered packets can cause LDA to incur a large error as we show in the evaluation. Intuitively, this is because a packet from the end of interval $u - 1$ may mix in to interval

u (via reordering) and the same interval may lose a packet sent at the end of interval u. The simple loss counters used in LDA cannot distinguish these two cases. The net effect is a serious discrepancy in the order of a measurement interval (*e.g.*, 1 second). Further, loss rate estimations using LDA do not work because LDA cannot distinguish reordered packets from lost packets. If the real loss is zero, LDA will false report alarms when there is only reordering. Recall that loss can cause TCP timeouts and latency violations; reordering across TCP flows cannot.

In order to address these shortcomings, we propose a new data structure called FineComb for fine-grain latency and loss measurements in the presence of reordering that we now explain.

3. FINECOMB

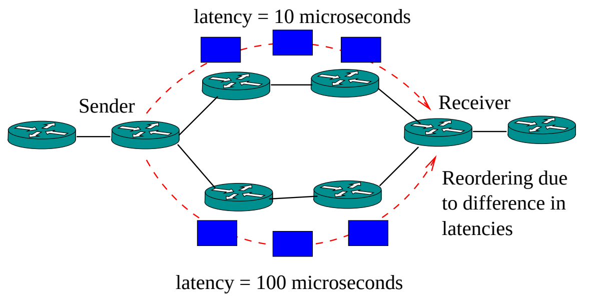
FineComb assumes a stream of packets going from a sender to a receiver. Time is divided into measurement intervals that are marked by interval start and end messages that are transmitted from the sender to the receiver. FineComb, like LDA, starts with the following simple idea. Suppose the sender and the receiver 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 the sender and the timestamp accumulator at the receiver, divided by the number of packets in the counter.

But how should the sender and receiver 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, the sender and receiver agree upon an interval of T packets that they would like to measure delays over. To do this, the sender marks off intervals by sending a special ‘sync’ control message each time it sends T packets to the receiver. (Note that the sender 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 sometimes refer to the first sync message in an interval as an interval-start message, and the end sync message as an interval-end message.

The challenge is that when packets traverse the network, they can arrive out of order as shown in Figure 2. The ordering of packets that are both transmitted and received within the interval itself 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 the sender, drift into another interval at the receiver. This situation is problematic



latency = 10 microseconds
Sender
Receiver
Reordering due to difference in latencies
latency = 100 microseconds

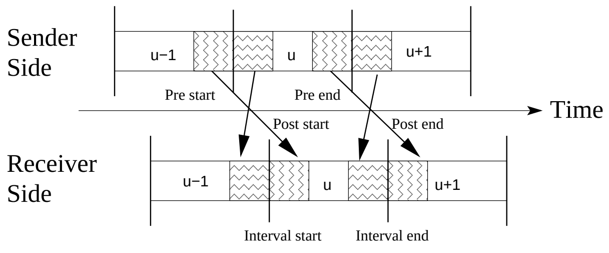


Figure 2: Four types of reordering that can occur.

since the timestamp accumulators at the sender and receiver 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 in the basic design of FineComb. First, packets sent at the end of interval $u-1$ can be routed on a high latency path and hence arrive at the receiver *after* the interval-start message. This can pollute interval u with extra packets; we call such packets *pre-start* packets. Second, packets from the start of interval u can be routed on a low latency path and hence arrive at the receiver before the interval-start message for interval u, so these *post-start* packets from interval u are effectively missing. Similar problematic reordering could also occur around the end of the interval (analogously referred to as *pre-end* and *post-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 misordering. This is because problematic reordering is confined to the reordering that occurs relative to the interval-start and interval-end messages. As shown at the top of Figure 2, this can occur if the sync messages are routed on one path (high or low latency) and the data messages following or preceding the sync message are sent on the other path. Thus $R \leq 2SL$, where S is the transmission speed and L is the maximum difference in latencies of paths. For example, if S is 10 Gbps and L = 100 μsecs and an average packet size of 1,000 bits, R is around 2,000 packets. By contrast, T may be as large as 5 million.

In addition to reordering, packets can also get dropped in the network, which can cause the sender and receiver state to become inconsistent. We assume at most βT packets from interval u will be dropped as they traverse the network from sender to receiver, 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 sender and receiver 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 sender and receiver 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 a consistent 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: Comparing counters at sender and receiver cannot be used to conclude that a bucket is useful in the presence of reordering because 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_2, ..., 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 [11]. However, since we are not operating in an adversarial setting, we can let $\odot$ will be a simple XOR operation, and we do not require the full power of a cryptographic hash function such as SHA-1; we can use simpler hash functions such as BOB [16] or H3 [25] instead.

The incremental stream digest has three useful properties. First, two streams containing different packets will hash to different values with high probability. Second, because XOR 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 XOR of their digest 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, this number 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 time-stamps. Thus the fact that R is much smaller than T is crucial to the efficiency of stash recovery.

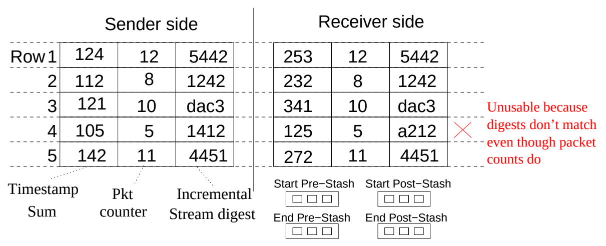
To show a concrete example of stash recovery, suppose a pre-start packet P from interval $u-1$ is hashed into the 20th bucket in interval u, making it useless. Assuming P is stored in the stash at the receiver because it arrived shortly after the interval-start message, stash recovery will look up the 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 a fixed memory budget M, 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 which measurements are computed is a key question that our later analysis will address.

3.3 Basic FineComb without a stash



	Sender side			Receiver side		
Row	Timestamp Sum	Pkt counter	Incremental Stream digest			
1	124	12	5442	253	12	5442
2	112	8	1242	232	8	1242
3	121	10	dac3	341	10	dac3
4	105	5	1412	125	5	a212
5	142	11	4451	272	11	4451

Unusable because digests don't match even though packet counts do
Start Pre–Stash
Start Post–Stash
End Pre–Stash
End Post–Stash

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

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 random 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:     $D \leftarrow compute\_hash(pkt)$          $\rightarrow$ Digest
3:     $i \leftarrow D \ mod \ M$          $\rightarrow$ Index into buckets
4:     $TS[i] \leftarrow TS[i] + \tau$
5:     $C[i] \leftarrow C[i] + 1$
6:     $D[i] \leftarrow D[i] \odot D$          $\rightarrow \odot$ could be XOR
7: 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:         $D \leftarrow D + (TS_r[i] - TS_s[i])$
5:         $N \leftarrow N + C_r[i]$
6:     end if
7: end for
8: $Average \ delay = D/N$
```

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 [10]. 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 [10]. Rather than waste 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 so 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.* post-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:         $L \leftarrow L + (C_s[i] - C_r[i])$
5:         $N \leftarrow N + C_s[i]$
6:     end if
7: end for
8: $loss \ rate = L/N$
```

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 pre-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 the evaluation section) 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 can 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. Notice that the pre-start stash and post-end stash contain sampled packets that potentially *drifted out* of the interval, and are thus missing from the receiver's buckets. Thus, these substashes can be used to *add* these packets back to the appropriate receiver's buckets. On the other hand, the post-start stash and pre-end stash contain sampled packets that potentially *drifted into* the interval. Thus, these substashes can be used to *subtract* these packets from the receiver's buckets.

Thus, stash recovery proceeds as follows. For each useless bucket, the receiver considers all the entries of the four substashes that map to that bucket. The receiver then considers *all subsets* 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 and receiver's incremental stream digest match for this subset of stash entries, then the receiver can recover that bucket by adding (if the entry is from the pre-start stash or post-end stash), or subtracting (if the packet is from the post-start stash or pre-end stash) the timestamps of those stash entries to the bucket's timestamp accumulator.

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 the next section) allow us to configure the sampling rate appropriately to ensure that optimal number of delay samples are obtained. In practice, we may not always know these values before hand, and they may change over time. One way to solve this problem, is to estimate the reordering and loss rate in online fashion, and use these estimates to compute the right sampling rates for future intervals. Unfortunately, there is no guarantee that reordering and loss exhibit any amount of consistency across intervals.

LDA also faces a similar problem—the loss rates are not clearly known a priori. To handle this case, LDA partitions resources into multiple banks, each bank statically 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. Thus, in FineComb, we use multiple banks optimized for the four different extreme operating regions: (β_{min}, ρ_{min}), (β_{min}, ρ_{max}), (β_{max}, ρ_{min}), and (β_{max}, ρ_{max}). Low values of β_{min} and ρ_{min}, means that the sampling rate chosen could be high, which in turn means the estimates are good. On the other hand, once the loss rate or reordering rate becomes high, this bank tuned for low loss rates may produce no valid delay or loss estimates, as the number of bad packets may far exceed the number of elements provisioned individually.

For the evaluation, we use the following simple (non-optimal) strategy to configure resources for each bank. We currently use four banks, each using one quarter of the total storage. We compute the optimal sampling probabilities and amount of stash required for each of the four extreme operating regions and partition resources statically. We leave dynamic reorganization of banks for future work.

For delay estimation (both mean and standard deviation), we compute the average delay output by all the individual banks. For loss, averages will not produce good output, especially at low loss rates. Instead, we use the following heuristic. 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

At the highest level, recall that main advantage of FineComb and LDA over the trivial algorithm that keeps a small sample of the timestamps of sent and received packets, is *sample efficiency*. If we can afford just 1,000 pieces of memory, the naive algorithm provides just 1,000 samples of delay. On the other hand, using aggregation allows the extraction of T samples (the number of packets sent in the interval, in the millions) if there is no loss. Since the standard error reduces by the square root of the number of samples, this can result in much finer delay estimates in the common case when the loss is small.

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 $S = M + W$.[1] Note that while we have formally proved the results in this section, for brevity, we only state the main theorems and results. (Proofs will be posted in the full version of this paper.)

4.1 Expected number of useful samples

Since our goal is to maximize 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 interval u, 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 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

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

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.[2] Thus, the expected number of bad packets that are sampled and not corrected is

$$E[L] = \beta pT + \max\{0, pR - W\} \quad (3)$$

Working with the conditional expectation. Because the distribution of L is quite complicated, in this section, we work with the conditional expectation $E\left[X|L = E[L]\right]$, which is obtained by plugging (3) 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, let's suppose we work with a fixed sampling rate p. In Appendix **??**, We can prove that the optimal size allocated to the stash is approximately:

$$W \approx \begin{cases} pR & \text{when } S \geq p(R+\beta T) \\ 0 & \text{else.} \end{cases} \quad (4)$$

Notice that (4) 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. Note that this analysis only applies to the optimal choice of stash size and sampling probability in order to calculate *latency*.

4.3 Optimizing sampling rate p.

No stash. Per (4) we now consider the case where we have no stash (*i.e.*, $W = 0$). In Appendix **??**, We can show that the optimal sampling rate is

$$p^{**} = \min\left\{\frac{S}{R+\beta T}, 1\right\} \quad (5)$$

[1]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.

[2]In practice, this may not be case; the stash may 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.

Stash. Now, (4) tells us that when we have a stash, its optimal size is $W = pR$. In Appendix **??**, We can show that when we use this value for the stash, the optimal sampling rate is approximately

$$p^* = \min \left\{ \frac{S}{2\rho^2 T} \left(2\rho + \beta - \sqrt{4\rho\beta + \beta^2} \right), 1 \right\} \quad (6)$$

where $\rho = R/T$.

To stash, or not to stash. The last issue we need to settle is whether it's 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 Appendix **??**) 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 $E[X]$ (see (**??**)) 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:

- How does FineComb perform in estimating mean delay, standard deviation and loss rates under different levels of reordering and loss rates? Does the number of effective samples obtained empirically agree with analytical bounds?

- How does an optimal configuration of FineComb compare with optimal configurations of other previous solutions such as LDA, in terms of relative error, under the assumption each solution is allowed to use the same total effective memory? (Here, we will assume that loss and reordering rates are known.)

- Since in practice, we cannot predict the loss and reordering rates, we consider FineComb provisioned with using multiple banks tuned to different loss and reordering rates. How does this perform compared to LDA ?

After describing the evaluation methodology we have used, we will discuss each of these three questions.

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 [18] so that fair comparison of FineComb with LDA is possible.

Delay model. Ideally, we would to use traces at two monitoring points within a real data center with GPS synchronized clocks; unfortunately, there exists no such publicly available data center latency traces. Prior work [18] used the Weibull delay distribution model empirically verified to mimic the distribution of delays within a backbone router by Papagiannaki *et al.* in [23]. We use mainly Weibull distribution (and Pareto for diversity) within our simulations. The delay for each packet is drawn is from a Weibull distribution, which has cumulative distribution function

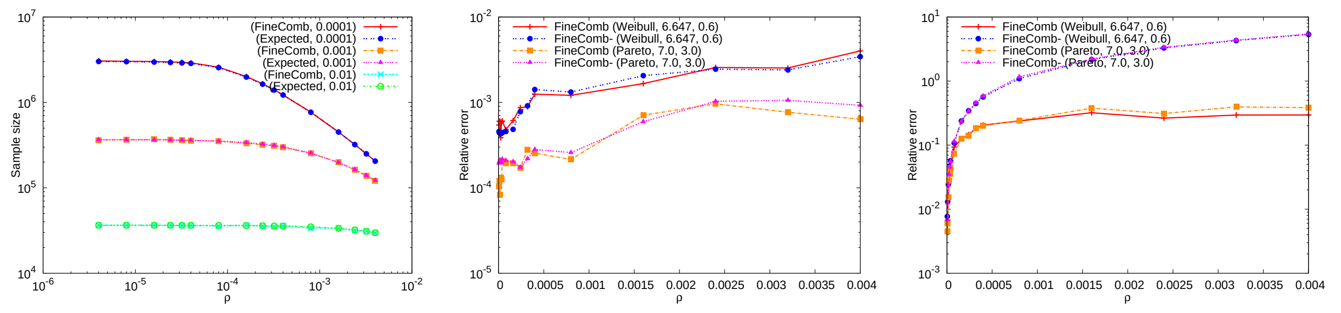
$$P(X \leq x) = 1 - e^{(-x/\alpha)^\beta}$$

with α and β representing the shape and scale of the graph respectively. We use [23]'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 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. Unless otherwise specified, 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.) Since the exact average latency is not as significant, we show our results in the form of relative error. On the other hand, the loss rate distribution is significant; we use Weibull 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, over a 10 Gbps bottleneck capacity with an inter-arrival time of 0.2μs—transmission time for 250 bytes at 10Gbps is $0.2\mu s$. (The average packet size or the inter-arrival time do not impact our results; the numbers are chosen similar to the experimental setting in [18], except for the higher path latencies in our setting.) Unless otherwise specified, our results are the average over 10 different simulation runs, taken in order to obtain smoother trends.

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 at all not 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



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

Figure 4: 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.

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 Section 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 Section 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 6 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 $<0, 0, x, 0>$ 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.

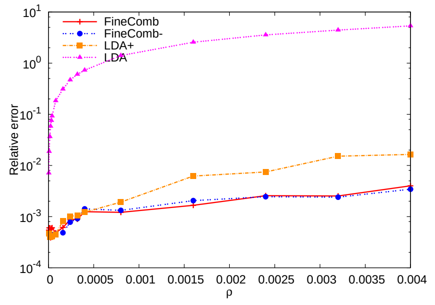
Resource configuration. We allocate a total of 1,000 buckets for FineComb. To simulate cases with and without stash, we assume stash elements are of the same size as bank elements (for simplicity). We use 64 bits from a 160-bit SHA-1 hash function for packet digests (other hash functions would work equally well). 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

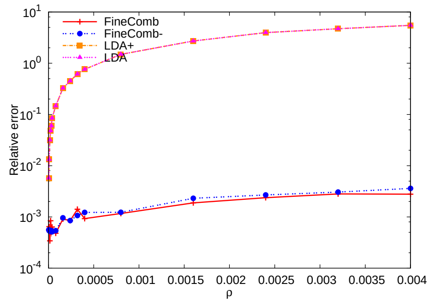
Expected number of samples. In our first experiment, we wish to understand how tight the theoretical bound on the number of useful samples is, at the optimal sampling probability. In Figure 4(a), we plot the expected number of samples according to the analytical bound given in Equation 2 (curve titled '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 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



(a) Mean delay, β = 0.0001, Forward



(b) Mean delay, β = 0.0001, Backward

Figure 5: Average relative error of mean delay estimates comparing FineComb with LDA with forward and backward reordering.

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 0.3% for either of the two distributions, under different levels of reordering.

As predicted by our theoretical work (in the full version), 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.

2. Sampled local timestamps. Similarly, consider the other trivial solution of sampling a small number of packets in each interval and storing their timestamps. 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 [18]) for all solutions.

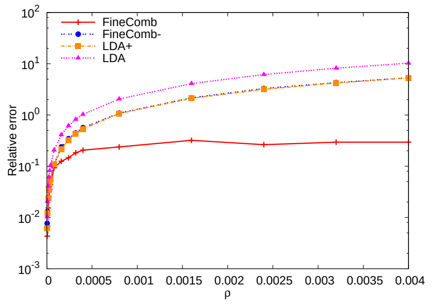
The main observation from the graphs is that, beyond small levels of reordering, LDA consistently performs the worst, with relative error as high as 100% (at $\rho = 0.0005$) to around 400% ($\rho = 0.004$). This follows from the fact that LDA cannot deal with reordered packets. If a reordered packet and a lost packet hash to the same LDA bin, the LDA will assume that bin is useful and include it in the loss rate 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). Further, this error is amortized over the total number of sampled packets (which gets progressively smaller as sampling rates are reduced to accommodate higher reordering).

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 cases, since in these cases, 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.

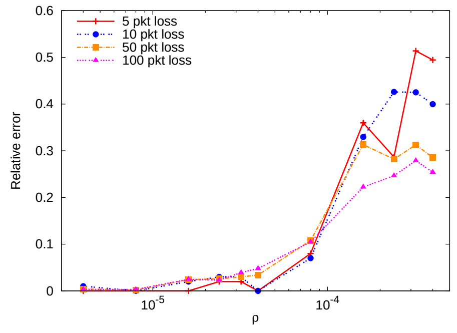
In all cases, we can observe that both FineComb and FineComb- perform consistently better than LDA even under high loss and reordering rates. We can observe that the relative error is mostly around 0.1% and never more than 1% in all the cases considered. For standard deviation estimates, we observed a similar phenomenon, *i.e.*, the accuracy of FineComb is orders of magnitude higher than LDA's. The same set of 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.

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.



(a) Low loss, β = 0.0001



(b) Microscopic loss, 5-100 lost packets

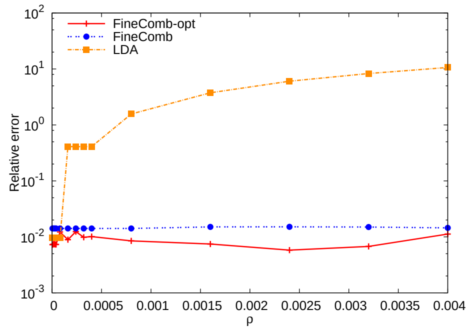
Figure 6: Relative error of FineComb at detecting low to microscopic loss rates.

5.4 Handling unknown loss and reordering rates

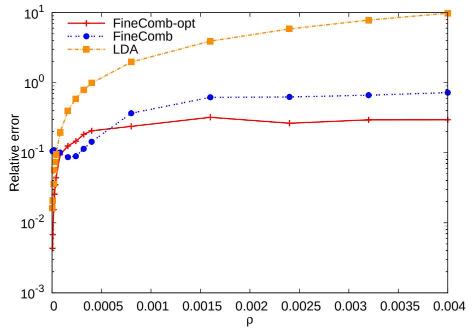
We have already demonstrated that it is easy to tune FineComb if the manager knows the loss and reordering rate. However, it is important to have a solution that works across a large range of loss rates and reordering rates using multi-bank FineComb.

We compare the efficacy of a 4-bank FineComb with a two-bank LDA. For FineComb, we optimize the individual banks for the four pair-wise combinations of $\beta_{min} = \rho_{min} = 0.0001$ and $\beta_{max} = \rho_{max} = 0.01$. In Figure 7(a), we show 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. We mainly show the relative error for mean delay for the worst case loss rate, when $\beta = 0.01$; the results for other loss rates, and for the backward reordering case, are quite similar to this curve (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 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.

In Figure 7(b), we show the relative error of the loss rate



(a) Mean delay estimation, β = 0.01, Forward



(b) Loss rate estimation, β = 0.0001, Forward

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.

estimation for the lowest injected loss rate situation (of 0.0001). We can observe a similar trend as we observed in the case of regular comparisons earlier, with FineComb's accuracy around 30% in all cases. The multibank variant's accuracy is comparatively worse at around 60%, but LDA is completely unusable.

6. IMPLEMENTATION

With 1000 buckets and 1000 stash entries, FineComb should take a small percentage of a low end 10mm$\times$10mm networking ASIC using a 400-MHz 65nm process. Key to a small footprint is a cheap version of an incremental stream digest using say an unrolled Rabin hash instead of the more expensive collision-resilient SHA-1 we used in simulation. A quicker path to deployment *today*, however, is using high-end FPGAs. For instance, we propose implementing FineComb on boards with Xilinx Virtex-4 FPGAs and 10-Gigabit Ethernet MACs that are currently available [1] for less than a thousand dollars per board (future NetFPGA boards [3] will also target 10 Gbps; the current version supports 4 Gbps and costs around $500). Alternately, boards containing network processors like the Intel IXP 2800, such as those used in the Open Network Laboratory [6] testbed, could be used to implement FineComb in software. For time synchronization, the boards need to have GPS chipsets (fairly cheap today), the solution used by monitors such as Corvil [2].

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 around 1.44, recovery takes no more than 14 msec. Note that it is not required that the processor be on-board, or that the board be directly connected to the the edge routers themselves; using span ports on routers/switches or using OpenFlow [7], one can mirror traffic to a PC with a monitoring board. Packet processing will be done in hardware on the board, but 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$\times$10 Gbps box [2]. 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. To be fair, the Corvil boxes allow visibility on a per-flow and application level but the difference in costs may make FineComb attractive for more widespread deployment. While implementing complete flow monitoring is unlikely to be feasible in a cheap monitoring board, there is often sufficient processing and memory in the boards; we propose to trigger finer-grain flow monitoring ("drill-down") when latency or loss violations are detected.

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 [29, 21, 28, 26]) and calculate the round-trip time of the packet. We have discussed the problems with active probes in Section 2.3. Router-based passive measurements is yet another active area of research [13, 14, 32, 24, 17]. 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.*, [12, 33]) where researchers proposed simple router extensions for latency measurements that are somewhat similar to the local timestamps idea discussed in Section 2.3, and hence share similar problems as indicated before.

Perhaps the most relevant research effort to ours is a recent data structure called LDA proposed by Kompella *et al.* in [18], and an incremental deployment architecture in [19]. Given the close similarity, we discussed it at length in the paper, and compared the performance of FineComb with LDA.

8. CONCLUSIONS

Measurement tools are badly needed to determine fine-grain latencies and losses that can affect application SLAs in data center environments. Many high-end data centers already employ boxes from vendors such as Corvil [2] and NetScout [4] at the edges. However, these solutions appear to be expensive and unscalable. There is a need for cheaper solutions that can be easily commoditized. While LDA does provide a commodity solution, it requires deployment on FIFO segments within each router and link, and does not extend directly to an edge-to-edge setting with persistent reordering.

This paper describes FineComb, an algorithm that can detect microsecond latency violations and loss as small as 1 in a million packets with a few thousand words of memory and simple logic. FineComb can be implemented in commodity boards that cost less than a few thousand dollars. FineComb introduces two new ideas, the addition of a incremental stream digest to 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.

To be fair, competing solutions have other benefits. For example, Corvil and NetScout provide visibility on a per-application basis. In contrast, LDA can provide fine-grain isolation of the router component causing latency violations if widely deployed. A combination could be effective in the marketplace. For example, boxes like Corvil could perhaps be made cheaper at 10 Gbps by implementing FineComb and only triggering fine-grain flow monitoring if performance violations are observed at the aggregate level. Meanwhile, fault-isolation would be greatly aided if LDA is gradually deployed. In data centers where seemingly imperceptible increases in latency have important business consequences, such monitoring tools can be a valuable investment.

Acknowledgements

The authors thank Kirill Levchenko, Michael Mitzenmacher and Zvika Brakerski for useful discussions.

9. REFERENCES

[1] 10-gigabit ethernet hardware demonstration platform. `http://www.xilinx.com/support/documentation/application_notes/xapp955.pdf`.
[2] Corvil tool minimises latency. `http://www.computerworlduk.com/technology/networking/networking/news/index.cfm?newsid=5797`.
[3] Netfpga. http://www.netfpga.org.
[4] Netscout. http://www.netscout.com.
[5] Next-generation routers: A comprehensive product analysis. http://www.heavyreading.com/details.asp?sku_id=662&skuitem_itemid=673&promo_code=&aff_code=&next_url=%2Fdefault.asp%3F.
[6] Open network laboratory. http://onl.arl.wustl.edu/.
[7] Openflow. http://www.openflow.org.
[8] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *SIGCOMM* (2008), pp. 63–74.
[9] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic flow scheduling for data center networks. In *USENIX/ACM NSDI* (Apr. 2010).
[10] 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.
[11] BELLARE, M., AND MICCIANCIO, D. A new paradigm for collision-free hashing: incrementality at reduced cost. In *In Eurocrypt97* (1997), Springer-Verlag, pp. 163–192.
[12] DUFFIELD, N. G., AND GROSSGLAUSER, M. Trajectory sampling for direct traffic observation. In *IEEE/ACM Transactions on Networking* (2000).
[13] ESTAN, C., KEYS, K., MOORE, D., AND VARGHESE, G. Building a Better NetFlow. In *ACM SIGCOMM* (2004), pp. 245–256.
[14] 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.
[15] GREENBERG, A. G., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. Vl2: a scalable and flexible data center network. In *SIGCOMM* (2009), pp. 51–62.
[16] JENKINS, B. Algorithm alley. Dr. Dobb's Journal, September 1997.
[17] KOMPELLA, R. R., AND ESTAN, C. The power of slicing in internet flow measurement. In *ACM/USENIX IMC* (May 2005).
[18] 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).
[19] KOMPELLA, R. R., SNOEREN, A. C., AND VARGHESE, G. mPlane: An architecture for scalable fault localization. In *ACM ReARCH* (2009).
[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] MARTIN, R. Wall street's quest to process data at the speed of light. `http://www.informationweek.com/news/infrastructure/showArticle.jhtml?articleID=199200297`.
[23] 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).
[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] SINHA, S., KANDULA, S., AND KATABI, D. Harnessing TCPs Burstiness using Flowlet Switching. In *HotNets* (San Diego, CA, November 2004).
[28] SOMMERS, J., BARFORD, P., DUFFIELD, N., AND RON, A. Improving accuracy in end-to-end packet loss measurement. In *SIGCOMM '05* (2005), pp. 157–168.
[29] SOMMERS, J., BARFORD, P., DUFFIELD, N., AND RON, A. Accurate and efficient SLA compliance monitoring. In *ACM SIGCOMM* (2007).
[30] 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 *Proc. ACM SIGCOMM* (Barcelona, Spain, Aug. 2009).
[31] WOVEN SYSTEMS, INC. EFX switch series overview. `http://www.wovensystems.com/pdfs/products/Woven_EFX_Series.pdf`, 2008.
[32] YUAN, L., CHUAH, C.-N., AND MOHAPATRA, P. ProgME: towards programmable network measurement. In *ACM SIGCOMM* (2007).
[33] ZSEBY, T., ZANDER, S., AND CARLE, G. Evaluation of building blocks for passive one-way-delay measurements. In *PAM* (2001).

APPENDIX

A. SETTING PARAMETERS.

A.1 Review of notations.

- Let T be the number of packets sent per interval.
- Let $R = \rho T$ be the number of problematically reordered packets in the interval.
- Let β be the fraction of dropped packets in the interval.
- Let $G \leq (1-\beta)T$ be the number of *good packets* that are received by the Receiver between the correct pair of 'Sync' messages.
- Let p be the FineComb sampling rate.
- Let M be the number of buckets in the FineComb.
- Let W be the number of entries in the stash.
- Let S be the total storage allocated to the FineComb with stashes.
- Let X be a random variable describing the number of useful samples extracted from FineComb.
- Let A be a random variable describing the number of *sampled* out-of-order packets in the interval that are *not corrected* during stash recovery.
- Let B be a random variable describing is the number of *sampled* dropped packets in the interval.
- Let $L = A + B$ be a random variable describing the number of *bad packets* that are *sampled* and *not corrected* during stash recovery.

A.2 $E[X|L]$: Conditional expected number of useful samples.

CLAIM A.1. *The conditional expected number of useful samples is* $E[X|L] = pG(1-\frac{1}{M})^L$.

PROOF. If we assume that all packets in the stream are distinct, then the L 'bad' sampled packets map to buckets of the LDA *independently* of the G 'good' packets. Then, it follows that the expected number of good samples from the LDA is [**?**]

$$\begin{aligned} E[X|L] &= E[\text{Good pkts / bucket}]E[\text{Number of useful buckets}] \\ &= \tfrac{p}{M}G(M - E[K]) \qquad (7) \end{aligned}$$

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

$$\Pr[K = k|L] = \frac{M!}{(M-k)!}\frac{S(L,k)}{M^L} \qquad (8)$$

where $S(L,k)$ is a Stirling number of the Second Kind. This claim follows by substituting $E[K|L]$ from the following Claim **??** into equation (**??**). □

CLAIM A.2. $E[K|L] = M(1-(1-\frac{1}{M})^L)$.

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

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

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

$$S(L,L) = 1 \qquad (11)$$

$$S(L,0) = 0 \qquad (12)$$

And now we begin:

$$\begin{aligned} E[K|L] &= \sum_{k=1}^{L} k\tfrac{M!}{(M-k)!}\tfrac{S(L,k)}{M^L} \\ &= \tfrac{1}{M^L}\sum_{k=1}^{L}\tfrac{M!}{(M-k)!}(S(L+1,k) - S(L,k-1)) \qquad (13) \end{aligned}$$

where we used (**??**). Now, using (**??**-**??**), we can find the first term of the sum as

$$\begin{aligned}\sum_{k=1}^{L} k \tfrac{M!}{(M-k)!} S(L+1,k) &= \sum_{k=1}^{L+1} k \tfrac{M!}{(M-k)!} S(L+1,k) \\ &\quad - \tfrac{M!}{(M-L-1)!} S(L+1,L+1) \\ &= M^{L+1} - \tfrac{M!}{(M-L-1)!}\end{aligned}$$

and the second term of the sum as

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

and plugging these back into (**??**) we get

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

as required. □

A.3 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 also 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

$$\Pr[A=a] = \begin{cases} \sum_{i=0}^{W} \binom{R}{i} p^i (1-p)^{R-i} & \text{when } a = 0 \\ \binom{R}{a+W} p^{a+W} (1-p)^{R-a-W} & \text{when } a \geq 1 \end{cases} \tag{14}$$

Since $L = A + B$, we can find the expectation of L as in (3).

A.4 $E[X]$: Unconditional expected number of useful samples.

We now derive combine the results of Claim **??**, and the distribution of L from Section **??** 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 $\mathcal{B}(p, \beta T)$ with $p \ll \beta T$), we

$$\begin{aligned}E[X|A] &= \sum_{b=0}^{\infty} E[X|A,b] \Pr[B=b] \\ &= \sum_{b=0}^{\infty} pG(1-\tfrac{1}{M})^{A+b} \cdot e^{-p\beta T} \frac{(p\beta T)^b}{b!} \\ &= pGe^{-\beta pT/M}(1-\tfrac{1}{M})^A \end{aligned} \tag{15}$$

We can also use the Poisson approximation for A to obtain

$$\begin{aligned}E[X] &= \sum_{a=0}^{\infty} E[X|a] \Pr[A=a] \\ &= pGe^{-\beta pT/M} \Big(\mathbf{F}(W; pR) \\ &\quad + \frac{e^{-pR/M}}{\left(1-\frac{1}{M}\right)^W} \left(1 - \mathbf{F}(W; (1-\tfrac{1}{M})pR)\right) \Big) \end{aligned} \tag{16}$$

where $\mathbf{F}(W; \lambda)$ is the cumulative Poisson distribution, that is

$$\mathbf{F}(W; \lambda) = \sum_{i=0}^{W} e^{-\lambda} \frac{\lambda^i}{i!} \tag{17}$$

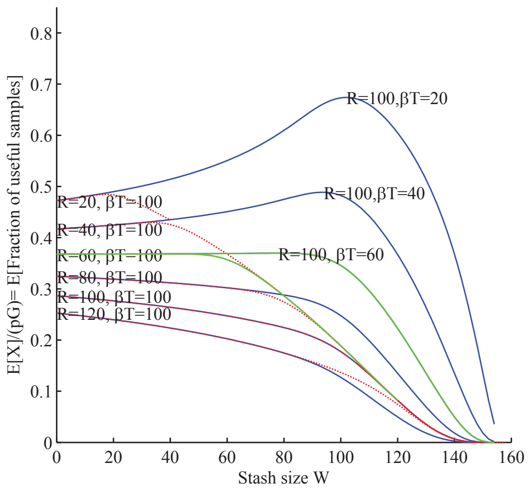


Figure 8: Expected fraction of useful samples $E[X]/pG$ vs stash size W when the total storage is $S = 160$. All the solid (blue) lines assume that the expected number of sampled out-of-order packets is $pR = 100$, and each line has a different value for the expected number of dropped packets $p\beta T$ 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 $p(R + \beta T) = S = 160$.

Since Equation (**??**) is so ugly, we will do most of our analytic work on

$$E[X|L = E[L]] = pG(1 - \tfrac{1}{M})^{\beta pT + \max\{0, pR - W\}} \tag{18}$$

and use numerical methods to work with $E[X]$ in (**??**).

A.5 Optimizing W for fixed sampling rate p.

We'd like to optimize the ratio between the LDA size and the stash size, using the fact that $S = M + W$ where S is fixed and sampling rate p is fixed.

Working with $E\left[X|L = E[L]\right]$. Substituting $M = S - W$ into (**??**), we obtain

$$E[X|L = E[L]] = pG(1 - \tfrac{1}{S-W})^{\beta pT + \max\{0, pR - W\}} \tag{19}$$

We optimize (**??**) in two different regimes. First, we assume

$$S \geq p(R + \beta T) \tag{20}$$

so that the total storage requirement for the FineComb and stash S is greater than the expected number of bad sampled packets. By plotting (**??**) in the regime of (**??**), we found that $E[X|L = E[L]]$ is maximized when $W = pR$. Second, we consider the regime where (**??**) does *not* hold, we find that $E[X|L = E[L]]$ is actually maximized when $W = 0$ so that $S = M$.

Working with $E[X]$. Now, we show qualitatively that the stash sizing in equation (4) obtained by working with the conditional expectation $E[X|L = E[L]]$ also applies when we work with the unconditional expectation $E[X]$ in (ugly) equation (**??**). To do this, we substitute $M = S - W$ into (**??**) and plot the resulting $E[X]$ as a function of W in Figure **??**.

From Figure **??** 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 of out 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 LDA, and set the stash size to $W = 0$. These qualitative results support our analysis using $E[X|L = E[L]]$, and so we present the results in (4).

A.6 Optimizing sampling rate p with no Stash.

CLAIM A.3. *When there is no stash, the optimal sampling rate is* $p^{**} = \frac{S}{R+\beta T}$

PROOF. We prove this directly from $E[X]$. When there is no stash, then $W = 0$ so that A in (**??**) is simply a binomial random variable $A \sim \mathcal{B}(p, R)$. Then, if put $M = S$ and approximate A as a Poisson random variable, the using (**??**) we find that the expected number of samples is

$$\begin{aligned} E[X|W = 0] &= \sum_{a=0}^{\infty} E[X|a] \Pr[A = a] \\ &= \sum_{a=0}^{\infty} pGe^{-p\beta T/S}(1 - \tfrac{1}{S})^a \cdot e^{-pR}\frac{(pR)^a}{a!} \\ &= pGe^{-p(R+\beta T)/S} \end{aligned} \tag{21}$$

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

A.7 Optimizing p with Stash $W = pR$.

Using $E[X|L = E[L]]$. Now, in (4) found that when we have a stash, it's 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 (**??**) to obtain

$$E[X|L = E[L], W = pR] = pG\left(1 - \frac{1}{S - pR}\right)^{\beta pT} \tag{22}$$

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

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

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

Working with $E[X]$. Now, we show qualitatively that the stash sizing in equation (4) obtained by working with the conditional expectation $E[X|L = E[L]]$ also applies when we work with the unconditional expectation $E[X]$ in (ugly) equation (**??**). 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 equation (**??**) as the dotted (red) lines in Figure **??**. For the second case, we assume that the stash is of size $W^* = pR$, and substitute $W = W^*$ and $M = S - W^*$ into (**??**) and plot the resulting as the solid (blue) lines in Figure **??**. We can make a number of observation from Figure **??**:

- **Stash is better than no stash.** From Figure **??** we can right away observe that for a fixed value for $R, \beta T$ 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 then the maxima of the dotted (red) curves).
- **Our approximation for p^* is good.** Furthermore, Figure **??** shows the our approximation for the optimal sampling rate p^* in (6) 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. Thus we arrive at our recommendations; we would like to operate the data structure at the maxima of the solid (blue) curves from Figure **??**. Thus, we suggest using a sampling rate of p^* per equation (6), and stash of size $W^* = 2\gamma p^* T$.

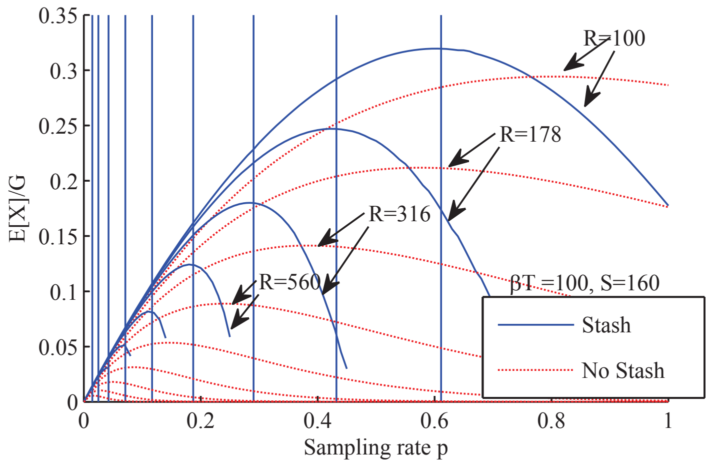


Figure 9: Expected fraction of useful packets $E[X]/G$ vs sampling rate p when the total storage is $S = 160$ and the number of dropped packets is $\beta T = 100$. The solid (blue) lines plot equation (??) when the stash is of size $W^* = E[A] = 2\gamma pT$, and for $M = S - W^*$ and the dotted (red) lines plot (??). Each pair of lines has a different value for the expected number of out-of-order packets $2\gamma T$ from the logarithmicaly-spaced set $2\gamma T \in \{100, 178, 316, 560, 1000, 1780, 3160, 5600, 10000\}$. The vertical lines are plotted according to equation (6) and represent the approximate maxima of the solid (blue) curves.