Unbiased Delay Measurement in the Data Plane

Yufei Zheng
yufei@cs.princeton.edu
Princeton University

Xiaoqi Chen
xiaoqic@cs.princeton.edu
Princeton University

Mark Braverman
mbraverm@cs.princeton.edu
Princeton University

Jennifer Rexford
jrex@cs.princeton.edu
Princeton University

Abstract

Network administrators are interested in continuously measuring the distribution of network delays, defined as the time between a request and its response. Ideally, such measurement can be done directly within the data plane of high-speed switches, where we can run succinct algorithms that process traffic at Terabits per second line rate. Unfortunately, measuring delays is challenging, given the considerable gap between the small available memory and the huge volume of arriving traffic. Existing data-plane algorithms for measuring delays exhibit bias against samples with higher delays, as these samples are more likely to be evicted due to hash collisions with new insertions. However, many important use cases, such as monitoring service-level agreements and video Quality-of-Experience, rely on accurate information about the tail of the delay distribution.

We present *fridges*, a novel data structure that produces *unbiased* estimates of the delay distribution, by correcting for the *survivorship bias* due to hash collisions in the data plane. We track the number of insertions into the data structure for each sample waiting in the fridge, and compensate accordingly when updating the delay distribution. We also show how to combine results from multiple fridges, each optimized for a different range of delays, for further accuracy gains. Simulation experiments show our design produces much better accuracy than prior work using naive hash-indexed arrays, achieving 2x-4x memory saving. We implement a prototype P4 program running on the Barefoot Tofino programmable switch, using only moderate hardware resources.

1 Introduction

Network administrators often want to measure fine-grained information about network delays, which directly impacts application performance. Delay can be characterized as the time difference between a *request* and its corresponding *response*, and can be calculated by observing the stream of network traffic packets. For example, the two-way delay for a TCP handshake is the time between a client sending TCP SYN packet to a server and receiving the corresponding TCP SYN/ACK packet, while the delay for a DNS lookup is the time between a DNS query packet and its corresponding reply packet. We can also measure the time between the beginning of a TCP flow (SYN packet) and its ending (FIN or RST packet), which characterizes the duration of a flow that corresponds to the delay an application experienced when downloading a file.

It is common for an Internet Service Provider (ISP) and its clients to specify a target delay distribution in their Service-Level Agreements (SLAs). For example, an ISP might be interested in identifying customers with more than 5% of TCP handshake delay in excess of 50 milliseconds.

However, unlike in a data-center network, an ISP might not be able to deploy measurement tools directly on customer end hosts; meanwhile, active measurement incurs overhead and may not accurately reflect the delay experienced by end-user applications. Thus, we often need to measure delay distributions by passively observing bidirectional application traffic at an ISP vantage point close to the customer hosts. By observing the stream of network packets, we can match an outgoing request packet with its incoming response packet, and subsequently calculate the pairwise delay on the internet and tally the delay distribution. It is also possible to separately calculate two legs of the delay (from client to the vantage point, and from the vantage point to server) and combine the measurements, as discussed in [10], to estimate the full client-to-server round trip delay.

The emergence of programmable data planes enables real-time monitoring directly in high-speed network devices. Switches and network interface cards with programmable data planes are now available off-the-shelf, and they are beginning to see commercial deployment in large data-center networks. We no longer need to mirror and store a large volume of traffic from the vantage point, and can instead implement customized algorithms to analyze traffic in the data plane and export summary statistics, such as heavy hitters [5, 9, 12], out-of-order packets [18], and so on. This not only helps reduce computation and network overhead, but also improves privacy by never exporting sensitive user traffic.

However, high-speed data planes have limited memory—not enough to store per-flow, let alone per-packet, state.

While it is possible to estimate volume-based metrics based on packet sampling or using memory-efficient sketch data structures, measuring delay requires matching information across *pairs* of packets, which imposes significant pressure on the memory from waiting for the second packet of the pair to arrive. This can easily be shown from a back-of-the-envelope calculation: Suppose we observe a backbone link running at a full line rate of 400 Gbps, which translates to around 2^{29} packets per second. With an average delay of 50ms, to store all such packets for the full time until the responses come back, we need an array with 2^{25} entries—at least an order of magnitude larger than the total data-plane register memory available in commodity programmable switches.

The programmable data plane poses several additional constraints beside the limited memory size. To maintain packet-processing speed at line rate, the data plane requires each packet to be processed within a constant number of clock cycles in a pipeline. Therefore, any operations on the data structure must finish within constant time. Meanwhile, we cannot perform arbitrary arithmetic operations, such as division or logarithm.

Due to these constraints, many earlier works rely on hash-indexed arrays to do operations in the data plane [3, 6, 20]. In this work, we also make use of hash-indexed arrays to match pairs of *request* and *response* to calculate delays. We assume each request and its matching response share an ID, which can be derived from header fields of the network packet. A naive solution based on [6] would save the request ID and a timestamp into an array, applying a hash function over the ID to obtain an array index. When the corresponding response arrives, we use the hash function again to look at the same index, and if the matching request exists, calculate this request's delay and update the statistics of delay distribution.

However, not all requests eventually receive a response. Problems arise when the memory is limited, as new requests would suffer from hash collisions with existing requests upon insertion. Upon hash collisions we cannot discard the new request and keep the existing one, otherwise the array would soon fill up with stale requests without a response [6]. But simply overwriting upon every hash collision is even worse: to produce a delay sample, the request must stay in the array for long enough without being overwritten. For large-delay pairs, the request needs to survive a large number of new insertions into the data structure without a hash collision. Consequently, more small-delay samples are produced, while large-delay pairs are undersampled and therefore *biased* against. This phenomenon is especially problematic in applications like verifying SLAs or measuring tail latency, since larger delays are exactly the anomalies we hope to catch.

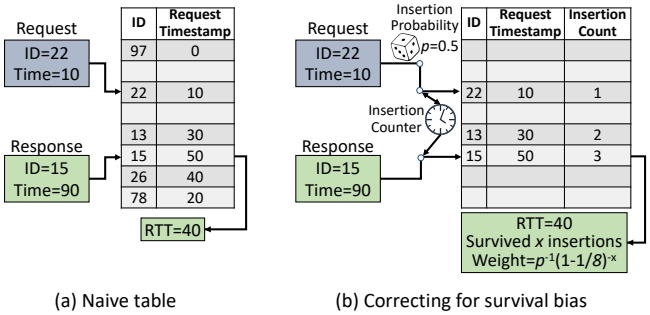

Figure 1: Compared with the naive approach, the fridge tracks how many insertions each entry survives, and assigns higher weights to less likely RTTs.

Previous efforts to mitigate the bias against larger delays include finding a middle ground between favoring the existing entries and overwriting aggressively. Chen et al. [6] used a large expiration threshold that corresponds to the 99th-percentile delay in the network, and only evicts upon hash collision when a record gets too old. Yet it is hard to accurately choose such a threshold, particularly when the prior distribution is unknown. Moreover, setting such a conservative threshold means stale requests stay too long in the memory, reducing the number of valid matches.

In this paper, we propose a data-plane algorithm that produces a provably unbiased delay distribution, specifically designed for programmable switches using the Protocol Independent Switch Architecture (PISA) [4]. Our algorithms tackle the bias by keeping track of the probability of getting each sample, and applying a correction factor inversely proportional to this probability when computing the distribution. This helps reduce the required memory size by 2x-4x compared with earlier work, while maintaining the same accuracy.

In what follows, § 2 describes the naive delay monitoring algorithm and why it is biased against high-delay samples, and § 3 introduces our design for *fridges*, a data structure that produces unbiased samples. In § 4, we discuss ways to extend our design beyond a single fridge. In § 5 we run simulation-based experiments for our algorithm and show it indeed estimates the delay distribution more accurately. § 6 presents and evaluates a prototype implementation of the fridge data structure on hardware programmable switches. We discuss related work in Section § 7, and conclude the paper in § 8.

2 Passive Delay Monitoring Problem

In this section, we first introduce the delay monitoring problem, and a "naive" data-plane solution. Then, we explain why

the naive algorithm is biased against large delays, and how we quantify the bias.

2.1 Naive delay monitoring

In delay monitoring, a stream of packets represents a stream of *requests* and *responses*, where we hope to match a response with its corresponding request, to generate useful delay statistics. Examples of request/response pairs include NTP request and response, DNS request and response over UDP, TCP handshake pairs (between SYN and SYN-ACK packets), and TCP flow duration (between TCP SYN and FIN/RST packets). In real-world applications, some requests never receive a response, for instance due to server failures or cyber attacks. We assume each request and its potential response carry the same identifier (ID) unique for the pair that allows matching, if the response exists. For example, we can extract IP addresses, port numbers, and sequence numbers from a TCP SYN packet, and match them with that of a TCP SYN-ACK packet. After matching a response to a request, the resulting delay sample contributes to some larger analysis of the delay distributions.

Today's programmable switches using the PISA pipeline architecture have tens of megabytes of register memory that we can read or write in the data plane, while processing individual network packets. To meet the strict speed requirements of line-rate packet processing (Terabits per second), the switch imposes several constraints on what packet-processing logic we can implement. The pipeline has a fixed number of stages, therefore each packet is processed within a constant number of clock cycles. To avoid memory hazard due to concurrent memory access, all register memory arrays are allocated to a specific pipeline stage, and we can only access one index of each register memory array when processing each packet.

Given the memory access constraints, we cannot implement traditional hash tables with sophisticated collision resolution logic. The *hash-indexed array* is a good candidate data structure to implement a naive algorithm for matching packet pairs and continuously generating delay samples, as illustrated in Figure 1(a).

(1) **Request:** For each request, we compute $hash(ID)$ to find an array index and write in the request ID and current timestamp. To avoid filling the memory with stale requests, we favor the new request upon hash collisions, by overwriting any existing request in the same array index.
(2) **Response:** For each response, we again compute $hash(ID)$ as the index. If there is a request with the same ID, we calculate the difference t of their timestamps, report an delay sample t, and remove the request; otherwise, there is no match and no sample is recorded.

2.2 Bias against large delays

The limited memory in the data plane poses a significant challenge. A higher delay means the request needs to stay in memory longer while waiting for its response to arrive, which translates to using more memory at any given point. Since we choose to always overwrite existing requests upon hash collisions, when memory is limited a request is more vulnerable to eviction the longer it remains in memory. This leads to a bias against larger delays.

At first glance, we could solve this problem by favoring *existing* requests on collisions. However, a response may never arrive, causing stale requests to consume the memory. An alternative is to set an expiration threshold and evict a record if the request stays in the data structure longer than the threshold. This method seemingly achieves a balance between overwriting aggressively and favoring existing requests conservatively, but in practice it is hard to find the right threshold [6]. A large threshold leads to an array full of stale requests, while a small threshold causes bias against larger delays, as such requests are quickly evicted before their responses arrive.

2.3 The delay distribution

In practice, we often hope to combine individual delay reports to generate some statistics of interest. In this work, we specify the output of our algorithms to be an approximated distribution of delays of all request/response pairs in the stream, and in particular we can look at the Cumulative Distribution Function (CDF) of the delay. We note that the estimation error of many real-world delay metrics commonly seen in Service Level Agreements (SLAs) can be translated to the difference between estimated and ground truth delay CDF curves. For example, the error in measuring 95th-percentile delay is the horizontal distance between the CDF curves at $y = 95\%$, while the error in measuring the fraction of RTTs above 40ms is the vertical distance between the CDF curves at $x = 40ms$. We therefore formalize the problem as follows:

Definition 2.1 (Delay problem). Given a stream of requests and responses with identifier ID_i, timestamp t_i, and type $c_i \in \{\texttt{req}, \texttt{resp}\}$ differentiating requests and responses, we pair a request i with its response i' when $ID_i = ID_{i'} \wedge (c_i, c_{i'}) = (\texttt{req}, \texttt{resp})$. Each request has a unique ID, and has at most one matching response. The *delay* of a request/response pair (i, i') is defined to be $t_{i'} - t_i$. Let $f(t)$ denote the number request/response pairs in the stream with delay t, and $F(t) = \frac{\sum_{\tau<t} f(\tau)}{\sum_t f(t)}$ the ground truth CDF. On seeing the entire stream, we hope to output a close approximation $\hat{F}(t)$ of $F(t)$.

It is also important to note that in this work we do not intend to distinguish between the delay CDF of all packets versus some subset of packets, e.g., those of one particular

flow or application. The algorithms presented in § 3 are general enough to be applied in either case.

3 Unbiased Delay Estimation

Contrary to previous work [6], we do not attempt to mitigate bias by fine-tuning the frequency of overwriting records. Instead, we overwrite aggressively to take in all new requests, and correct for bias as samples are collected. We start this section with the idea of bias correction (§ 3.1). Next we describe the single *fridge* algorithm and its mechanism of applying correction factors (§ 3.2).

3.1 Correcting for survivorship bias

The phenomenon that larger delays are undersampled traces back to the step where any report, whether the delay is large or small, is considered as one sample. Instead, we should cherish each sample with a high delay—it should account for not only itself, but other sibling requests that are evicted before their responses arrived.

Thus, we define a correction factor to counter this survivorship bias. If a sample has a probability q to survive without being evicted, upon seeing its report we set the correction factor to $\frac{1}{q}$, and count it as $\frac{1}{q}$ samples. This way, evictions from the data structure no longer lead to biases, since the collected large-delay reports can compensate for the missing ones.

To track a sample's survival probability, we can analyze the number of insertions between the time this sample's request is inserted and its response arrives. Each insertion has a small probability to evict the request due to hash collision, thus the sample's survival probability diminishes as there are more insertions before the response.

3.2 Single fridge algorithm

Now we discuss the design of a single *fridge* data structure. We precisely track the "time" each request stayed in the fridge and calculate the correction factor, as illustrated in Figure 1(b). This is inspired by humans checking how long a grocery item has stayed in a fridge when taking it out.

We formally compute the correction factor using the inverse of the probability of a sample being collected. We define a data structure, *fridge*-(M, p), to be a hash-indexed array of size M equipped with an entering probability p, which dictates that each new request is inserted into the array with probability p (and discarded with probability $1-p$).

3.2.1 Probability of Survival. A request's probability of survival decreases as it spends more "time" in the fridge, measured by the number of other requests being inserted between this request and its response. We track this number by maintaining a global insertion counter. For each new request, an existing request in the fridge has a $\frac{p}{M}$ chance to suffer from a hash collision and be evicted; thus requests stay for roughly $\frac{M}{p}$ insertions, which we call the *average lifetime* of a fridge.

Consider a sample with delay t that survives x insertions between its request and response: it must survive three independent events: (1) its request enters the fridge (with probability p), (2) the request survives next x insertions into the array with size M, and (3) the sample has delay t in the underlying ground truth distribution. Denote $f(t)$ as the true number of samples with delay t, and n the true number of samples in the stream, we have

$$\mathbb{P}[\text{getting a sample with delay } t] = p \cdot \left(1-\frac{p}{M}\right)^x \cdot \frac{f(t)}{n}. \quad (1)$$

Note that the first two terms indicates a sample's survival probability $q = p \cdot \left(1-\frac{p}{M}\right)^x$. We therefore set a correction factor of $\frac{1}{q} = p^{-1} \cdot \left(1-\frac{p}{M}\right)^{-x}$ for each report we observe. After seeing the entire stream, we get a collection of reports, where each contains its delay t_i and its correction factor, in the form of a 2-tuple: $\left(t_i, p^{-1}\left(1-\frac{p}{M}\right)^{-x_i}\right)$, with x_i being the number of insertions the sample survives. From the reports we can obtain an estimator of $f(t)$ for all t, denoted as $\hat{f}(t)$, by summing up all correction factors that correspond to t.

We show in Lemma 3.1 that $\hat{f}(t)$ is an unbiased estimator of $f(t)$ with bounded variance.

Lemma 3.1. *Let Y_i be in indicator of sample i, $Y_i = 1$ if sample i has delay t, and $Y_i = 0$ otherwise, then*

$$\hat{f}(t) := \sum_{i\in[n]} p^{-1}\left(1-\frac{p}{M}\right)^{-x_i} Y_i$$

is an unbiased estimator of $f(t)$, and

$$\mathrm{Var}[\hat{f}(t)] = \frac{f(t)}{n}\sum_{i\in[n]}\left(p_1^{-1}\left(1-\frac{p_1}{M}\right)^{-x_i} - \frac{f(t)}{n}\right).$$

Proof. Fix any t, and from Eq. 1,

$$\mathbb{E}[Y_i] = \mathbb{P}[\text{get a sample with delay } t] = p\left(1-\frac{p}{M}\right)^{x_i}\frac{f(t)}{n},$$

$$\mathrm{Var}[Y_i] = p\left(1-\frac{p}{M}\right)^{x_i}\frac{f(t)}{n}\left(1-p\left(1-\frac{p}{M}\right)^{x_i}\frac{f(t)}{n}\right).$$

By definition, $\hat{f}(t) = \sum_{i\in[n]} p^{-1}\left(1-\frac{p}{M}\right)^{-x_i} Y_i$, then

$$\mathbb{E}[\hat{f}(t)] = \sum_{i\in[n]} p^{-1}\left(1-\frac{p}{M}\right)^{-x_i}\mathbb{E}[Y_i] = f(t).$$

Since indicator Y_i's are independent,

$$\begin{aligned}\mathrm{Var}[\hat{f}(t)] &= \sum_{i\in[n]} p^{-2}\left(1-\frac{p}{M}\right)^{-2x_i}\mathrm{Var}[Y_i] \\ &= \frac{f(t)}{n}\sum_{i\in[n]}\left(p^{-1}\left(1-\frac{p}{M}\right)^{-x_i} - \frac{f(t)}{n}\right). \quad (2)\end{aligned}$$

□

3.2.2 Approximated CDF. We estimate the CDF $\hat{F}$ by aggregating and normalizing the individual point-wise estimate $\hat{f}$

using discrete integration,

$$\hat{F}(t) := \sum_{i,t_i \le t} \frac{\hat{f}(t_i)}{\sum_i \hat{f}(t_i)}, \forall t \in \{t_i\}_i. \quad (3)$$

To compare $\hat{F}$ with the ground truth CDF F, it is convenient to think of $\hat{F}$ and F as continuous functions. Throughout this work, we assume linear interpolation for both $\hat{F}$ and F.

Though $\hat{f}(t_i)$ is unbiased, and so is $\sum_i \hat{f}(t_i)$ by linearity of expectation, it is important to note that the unbiasedness is not preserved under division. Therefore, $\hat{F}$ is not necessarily unbiased. Nonetheless, the single fridge algorithm improves on the naive algorithm considerably in terms of reducing bias (§ 5.1). Later in § 5.2, we show how the choice of p affects accuracy of the estimated CDF and how to identify a good p based on the fridge size and the maximum delay.

3.3 Setting the Tunable Parameters

Choosing the entry probability (p). Given a fixed M, the entry probability p needs to be strategically chosen such that the (M, p)-fridge operates optimally. Recall that on average each request survives M/p insertions (the "average lifetime" of the fridge) before being evicted.

- If delays are short such that most request-response pairs we are measuring are less than M/p insertions apart, almost all responses will arrive before any collision happens to the request. For most samples, the insertion counter x will be 0, therefore the correction factor is always p^{-1} with no effect of the survivorship bias. In short, memory is "underutilized".
- If the average lifetime M/p is too short (there are often more than M/p insertions between request-response pairs), most requests cannot survive long enough until their responses arrive – a request will suffer from hash collisions with overwhelming probability. The very lucky requests that did survive will have very large correction factors, leading to an enormous estimation variance. Although our estimation is still unbiased in this case, it has little practical value. Under this scenario, memory is "oversubscribed".
- Ideally, the fridge operates in the regime with its average lifetime M/p aligning with the number of insertions between most request-response pairs. In this case, x (the number of insertions survived by a sample) is close to M/p.

We note that in the ideal regime, given a large M and the assumption $x \approx M/p$, typical delay samples should have a correction factor in the order of $p^{-1}\left(1 - \frac{p}{M}\right)^{-M/p} \approx e/p$. Thus, medium-delay samples weights about e times more than samples with very short delay.

Inferring the number of insertions (x). We further observe that, with a constant traffic rate, the number of insertions survived by a sample (x) is proportional to the delay observed by this sample. Therefore, it is possible to not record an insertion counter in the fridge, and use the delay to recover x and calculate the correction factor. However, in many network applications, delay is correlated with short-term spikes in traffic rate (which cause transient congestion), which are precisely the anomalous events we want to scrutinize. Thus, we still opt to record the exact insertion counters.

Regarding the number of table entries (M). We note that although we expect a fridge to have thousands of entries in practice, in the extreme case we require $M \ge 2$, as we account for each survived request's survivorship bias using the collisions suffered by its $M - 1$ sibling requests. A single-entry fridge with $M = 1$ is a degenerative case, as we cannot observe any "survivorship bias" (all samples have $x = 1$). In the special case of $M = 2$, the surviver's correction factor doubles every time its sole sibling is replaced due to a new insertion. This estimation clearly has a large variance, and a larger M is preferred — the estimation about survivalship bias becomes more accurate as we observe the fate of more sibling requests.

As the fridge size M is limited by the memory available under the given hardware environment, given a fixed M it is important to provision the fridge with the appropriate entry probability p, based on the traffic rate (number of requests per second) and the delay we expect to observe. In § 5.2 we evaluate the effect of choosing p on a fridge's accuracy, and demonstrate the fridge has some tolerance regarding this choice.

4 Expanding Beyond a Single Fridge

In this section, we discuss how to extend the single fridge design to possibly achieve better measurement accuracy. We first discuss why a simple design using multiple hardware pipeline stages to build a single fridge will, surprisingly, hurt high-delay samples (§ 4.1). Then, we present how to build multiple fridges to and combine their output correctly (§ 4.2).

4.1 Using many pipeline stages per fridge

On a PISA programmable switch [4], we are limited to accessing only one index per register array when processing a packet. Algorithms often span multiple pipeline stages and allocate multiple register arrays to improve performance. For example, the delay measurement algorithm in [6] achieves the best accuracy when the same total memory size is split into 4-6 arrays in separate pipeline stages, each indexed with a different hash function. Naturally, when running the fridge algorithm, we should also consider a multi-stage design.

Similar to [6], we could use multiple memory arrays indexed by different hash functions.This makes hash collisions independent between different arrays, and generally performs better in many data structure use cases, thanks to the "power of two choices" phenomenon. However, to deal with stale requests, upon a hash collision we must favor inserting the new request and evict the existing request. We could implement a scheme similar to HashPipe [20], where the request evicted in the first stage is inserted again in the second stage, and likewise for later stages. The propagation stops when an empty array slot is encountered, and a request is finally abandoned when it is evicted from the very last pipeline stage.

To produce an unbiased delay estimator, we need to find the right correction factor under this design. We note that a request in the first few stages is not at risk of eviction, thus their survival has probability one; only the requests appearing in the last stage need a correction factor for its survival probability, based on x_i, the number of insertions it survived in the last stage.

Unfortunately, this design works poorly for samples with larger delay. Assume we build a D-stage fridge with total memory M and entry probability p, we can calculate the number of insertions a request can survive in the fridge as a probability distribution. Note that each stage has M/D entries, and for simplicity we assume the memory is full of requests with no empty slots. For one insertion, a request currently in stage s has probability $p \cdot D/M$ suffering from a collision and move to stage $s+1$. Thus, the number of insertions a request survives in stage s follows the geometric distribution $l_s = Geo(p \cdot D/M)$. We can thus write the lifetime distribution of the D-stage fridge, i.e., the total number of insertions survived before a request is evicted from the last stage, as $L_D = \sum_{s=1}^{D} l_s = \sum_{s=1}^{D} Geo(p \cdot D/M)$, with expectation $D \cdot \frac{M}{p \cdot D} = M/p$. Meanwhile, for an ordinary fridge, we can simply plug in $D = 1$: its lifetime distribution is simply $Geo(p/M)$ with expectation M/p.

Although items in the multi-stage fridge have the same expected lifetime $\frac{M}{p}$ as those in a single-stage fridge, its lifetime distribution is the sum of D i.i.d. geometric variables, which is more concentrated and has a lighter tail than a single geometric variable. This is to say, for a large delay $T > M/p$ and $D > 1$, we have

$$\mathbb{P}[Geo(p/M) \geq T] > \mathbb{P}[\sum_{s=1}^{D} Geo(p \cdot D/M) \geq T]. \quad (4)$$

Therefore, requests with delay higher than the fridge's average lifetime has a much smaller survival probability.

This phenomenon is most obvious when we consider the extreme case: with $D = M$ stages each having array size 1, the fridge essentially becomes a FIFO queue, and the lifetime distribution becomes very narrowly concentrated. With every request spending almost the same time in fridge, a request whose delay is higher than the average lifetime has almost no chance to survive. Instead, we want the exact opposite: the lifetime distribution should be heavy-tailed, so requests have some probability of staying in the fridge for much longer than $\frac{M}{p}$ insertions, so our fridge can collect some samples for large delay. Thus, analytically the multi-staged design performs poorly; we have also verified this phenomenon empirically.

Thus, we should never use a multi-stage fridge. When we need to utilize more memory than the capacity of a single pipeline stage, we should simply merge the memory across multiple stages into one large logical hash-indexed array and build a single-stage fridge. We also note that proposals like dRMT [7] would enable stateful memory allocation across stages, so a simple one-stage algorithm can use the entire stateful memory directly.

4.2 Using multiple fridges

Requests in one fridge have an average lifetime $\frac{M}{p}$, which can be adjusted to fit the typical delay of the input traffic stream for higher accuracy. However, internet traffic exhibits a wide range of delays, due to different geographic distances, server behavior, and congestion conditions. Thus, a single fridge targeting a particular $\frac{M}{p}$ may be inadequate.

To cover a wide range of delays, we split the memory into N fridges with size $M_1 + \cdots + M_N = M$. Requests and responses are directed to one of the fridges via a hash function, while each fridge has its own entry probability $p_1 + \cdots + p_N < 1$ and targets a different average lifetime $M_1/p_1, \ldots, M_N/p_N$. When a response matches in fridge k, we calculate the correction factor $p_k^{-1}(1 - p_k/M_k)^{-x_i}$ based on fridge k's entering probability p_k and the probability for surviving x_i insertions in this fridge.

Since different fridges have different average lifetime, they have different variance when estimating various ranges in the delay distribution. We need to combine their output strategically to produce the final estimated delay distribution with minimum estimation variance. In § 5.3, we demonstrate that using multiple fridges can indeed produce more accurate estimate than a single fridge when memory size is limited.

We now describe the process of combining multiple fridge's output using the Inversed Variance Weighting method [8] in more detail.

Variance of each fridge. For a sample coming from fridge k (with size M_k and entry probability p_k) that survives x insertions, we set its correction factor as $p_k^{-1}\left(1 - \frac{p_k}{M_k}\right)^{-x}$ following Lemma 3.1. Summing up all correction factors for a t coming out of fridge k gives an unbiased estimator $\hat{f}_k(t)$ for $f(t)$, the true number of samples with delay t, where the

unbiasedness follows again from Lemma 3.1. Let k_i be the index of the fridge sample i comes from, then the variance of $\hat{f}_k(t)$ follows directly from Eq. 2,

$$\mathrm{Var}[\hat{f}_k(t)] = \frac{f(t)}{n} \sum_{i\in[n],k_i=k} \left(p_k^{-1}\left(1-\frac{p_k}{M_k}\right)^{-x_i} - \frac{f(t)}{n}\right). \tag{5}$$

The weighted average of estimators. Similar to classical sketching algorithms such as CountMin [9] and CountSketch [5], in the multi-fridge algorithm, we keep a set of N basic unbiased estimators $\{\hat{f}_1(t), \hat{f}_2(t), \ldots, \hat{f}_N(t)\}$ for $f(t)$, each coming from a fridge. Since the variance of each individual estimator could be large, we combine them to get a better estimator. However, unlike [5, 9], our basic estimators have different variance, so instead of simply taking their min or the median, we leverage this fact to compute a weighted average of the estimators.

It is well-known in statistics [8] that given N unbiased estimators with bounded variance, we can set weights optimally so that the weighted average of these estimators has the minimum possible variance.

Theorem 4.1. *Given N unbiased estimators $\hat{f}_1, \hat{f}_2, \ldots, \hat{f}_N$ with bounded variance $\mathrm{Var}[\hat{f}_1], \mathrm{Var}[\hat{f}_2], \ldots, \mathrm{Var}[\hat{f}_N]$ respectively, the set of N weights $\{w_1, w_2, \ldots, w_N\}$ with $w_k = \frac{\frac{1}{\mathrm{Var}[\hat{f}_k]}}{\sum_k \frac{1}{\mathrm{Var}[\hat{f}_k]}}$, $k \in [N]$ minimizes the variance of the combined unbiased estimator $\sum_k w_k \hat{f}_k$.*

To combine estimators using Theorem 4.1, we need to associate each sample with a weight, so as to calculate weights $\{w_1, w_2, \ldots, w_N\}$ for a fixed delay t. Therefore, the report sample i in the multi-fridge algorithm becomes a 4-tuple that keeps the weight of the sample, to be determined next, as well as the fridge index k_i, on top of the delay t_i and the correction factor as in the single fridge case.

However, Theorem 4.1 cannot be directly cast into our multi-fridge setup, since we do not know variance $\mathrm{Var}[\hat{f}_1]$, $\mathrm{Var}[\hat{f}_2], \ldots, \mathrm{Var}[\hat{f}_N]$ exactly. We work around this issue by approximating weights w_k for each fridge k. From Eq. 5, we can safely focus on estimating

$$\frac{f(t)}{n} \sum_{i\in[n],k_i=k} p_k^{-1}\left(1-\frac{p_k}{M_k}\right)^{-x_i} \tag{6}$$

in the variance, since $\frac{f(t)}{n} << 1 << p_k^{-1}\left(1-\frac{p_k}{M_k}\right)^{-x_i}$. Yet, it would be false to assume the $\frac{f(t)}{n}$ factor outside of the summation cancels out in w_k, so we can obtain the rest of (6) precisely from summing over correction factors from all reports. This would have produced an underestimation, since n is the true number of samples, and the fridges can only report fewer than n samples due to hash collisions.

We therefore use the unbiased estimator of (6),

$$\sum_{i\in[n]} p_k^{-2}\left(1-\frac{p_k}{M_k}\right)^{-2x_i} Z_i, \tag{7}$$

where indicator $Z_i = 1$ if sample i comes from fridge k and has delay t, and $Z_i = 0$ otherwise. An argument similar to that in Lemma 3.1 suffices to verify the unbiasedness of (7).

Putting all elements together, the report of a sample with delay t that survives x insertions in fridge k, the 4-tuple (fridge index, delay, correction factor, weight factor), has the following form

$$\left(k, t, p_k^{-1}\left(1-\frac{p_k}{M_k}\right)^{-x}, p_k^{-2}\left(1-\frac{p_k}{M_k}\right)^{-2x}\right).$$

We obtain estimator $\hat{f}_k(t)$ by summing over all correction factors of samples with RTT t from fridge k,

$$\hat{f}_k(t) := \sum_{i\in[n]} p_k^{-1}\left(1-\frac{p_k}{M_k}\right)^{-x_i} Z_i.$$

Denote the unbiased estimator of (6) as $\hat{V}_k(t)$, by (7),

$$\hat{V}_k(t) := \sum_{i\in[n]} p_k^{-2}\left(1-\frac{p_k}{M_k}\right)^{-2x_i} Z_i.$$

Finally, we obtain our multi-fridge estimator $\hat{f}(t)$ through a weighted average of estimators from all fridges $\{\hat{f}_1(t), \hat{f}_2(t), \ldots, \hat{f}_N(t)\}$,

$$\hat{f}(t) := \sum_k \hat{w}_k(t)\hat{f}_k(t), \text{ where } \hat{w}_k(t) = \frac{\frac{1}{\hat{V}_k(t)}}{\sum_k \frac{1}{\hat{V}_k(t)}}.$$

This concludes the process of combining the output of multiple fridges. Note that despite of the approximation, we always have $\sum_{k\in[N]} \hat{w}_k(t) = 1$, and $\hat{f}(t)$ is hence unbiased for being a convex combination of unbiased estimators.

5 Evaluation

In this section, we use real-world and synthetic traffic traces to show that the fridge algorithm can effectively reduce bias in delay measurement, compared with prior works. To experiment with different parameter settings, we run all tests using a Python-based simulator. We discuss and evaluate a prototype running on hardware programmable switches in § 6.

Distance metric. We evaluate the accuracy of single- and multi-fridge algorithms by computing the distance between the ground truth CDF $F(t)$ and the estimated CDF $\hat{F}(t)$ computed by our algorithms. As discussed in § 2.2, the CDF is closely related to criteria specified in SLAs. For example, "95th-percentile delay" is where the delay CDF curve crosses $y = 95\%$. Since real-world delays vary widely, absolute error is not an effective metric; we instead look at the relative error of percentile delay queries: $\left|\log_2\left(\frac{\text{Estimated}}{\text{Ground Truth}}\right)\right|$, which corresponds to the horizontal distance between the estimated and ground truth CDF curve under logarithmic x-axis. We are

interested in the relative error of typical percentile queries (at 50%, 95%, and 99%), as well as the maximum error for any percentile between [5%, 95%], i.e., the maximum horizontal gap between the CDF curves between $y \in [5\%, 95\%]$.

Dataset. We use both real and synthetic network traffic traces in our experiments.

- Real-world traffic (§ 5.1, 5.3): We use a bi-directional anonymized traffic trace that contains 10 million packets across 11.4 seconds, collected from a 10Gbps border link of a local ISP network. We extract round-trip delay samples by treating outgoing TCP data packets as requests, and looking for their matching incoming TCP acknowledgment packets as responses. The trace includes 61% requests and 39% responses. Approximately 13% of all requests has a matching response, as the TCP delayed-ACK mechanism only sends one response for every two requests, and malicious traffic such as port-scanning attacks generates many orphan requests. The average round-trip delay across all samples is 57.8 milliseconds.
- Synthetic trace (§ 5.2): We also generated synthetic traces to explore our data structure's performance characteristics under other traffic distributions. We first generate request packets arriving at a constant rate of 1 million packets per second, and randomly select a 40% subset to generate responses. Subsequently, given a maximum delay of Tms, we randomly sample an delay from a log-uniform distribution between $(0, T)$ms for each response. Finally, we combine the requests and responses and sort them by their timestamps. The trace contains 0.5 million delay samples, with approximately 1.75 million packets in total. Although the synthetic trace is not fully realistic, it allows us to test our data structure by changing the delay distribution.

Unless otherwise noted, we repeat each experiment ten times with different hash seeds and combine estimated CDFs, to reduce variance from individual runs and highlight the bias. We note that our algorithm's output exhibits a similar variance comparable to [6]; the output distribution is almost the same across different runs, unless memory is extremely limited.

5.1 Comparison with naive algorithm

We first show that our fridges achieve higher accuracy than the naive algorithm described in § 2.1.

In Figure 2, we visualize the advantage of the fridge algorithm by plotting the estimated delay CDF curves alongside the ground truth (shaded). We run the single-fridge algorithm using entry probability $p = 1$ and memory size $M = 2^{16}$, and the naive algorithm in [6] using the same memory size on the real-world traffic trace. The naive algorithm uses an expiry

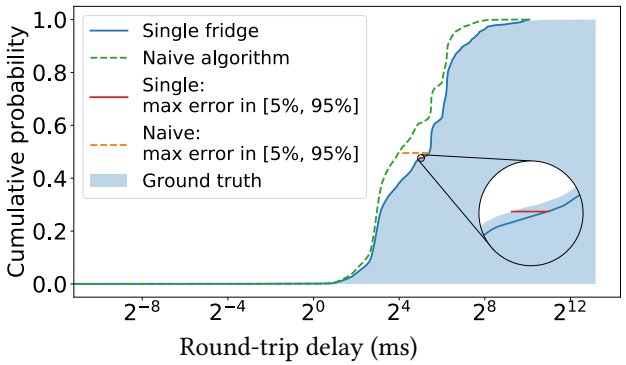

Figure 2: (M=2^{16},p=1)-fridge produces a visibly more accurate delay CDF, compared with the naive algorithm using the same memory size M=2^{16} and expiration threshold T=2^9ms. The fridge and the naive algorithm achieves maximum relative error of 8% and 168% respectively for percentile delay queries.

threshold $T = 2^9$ms, close to the 99%-percentile delay in the ground truth, as suggested by the authors of [6].

As we can see from Figure 2, our unbiased fridge algorithm closely reproduces the ground truth CDF curve. The naive algorithm produces a CDF curve biased against high delays, underestimating percentile delay queries.

The fridge algorithm estimates 50th, 95th, and 99th percentile delay much more accurately than the naive algorithm. We also plotted the maximum horizontal gap between estimated and ground truth CDF curves in [5%, 95%], which corresponds to the maximum relative error when answering percentile delay queries for any percentile between 5% and 95%. The fridge algorithm has a maximum relative error of 8%, while the naive algorithm has a maximum relative error of 168%. We observe similar results when using synthetic traces.

5.2 Choosing the best entry probability

Requests in a (M, p)-fridge have an average lifetime of M/p insertions. Although the fridge algorithm's output is guaranteed to always be unbiased, we can improve its accuracy by choosing p carefully to reduce the estimator's variance. In general, we want more samples to reduce the variance of the fridge's estimation. When memory is limited, a higher p shrinks the average lifetime, so fewer large-delay samples can survive; however, a very small p means not many requests enter the fridge in the first place, thus it cannot produce many samples either.

As high-delay samples are the hardest to measure, intuitively, the best p that maximizes accuracy should make the fridge's average lifetime ($\frac{M}{p}$ insertions) roughly equal to the

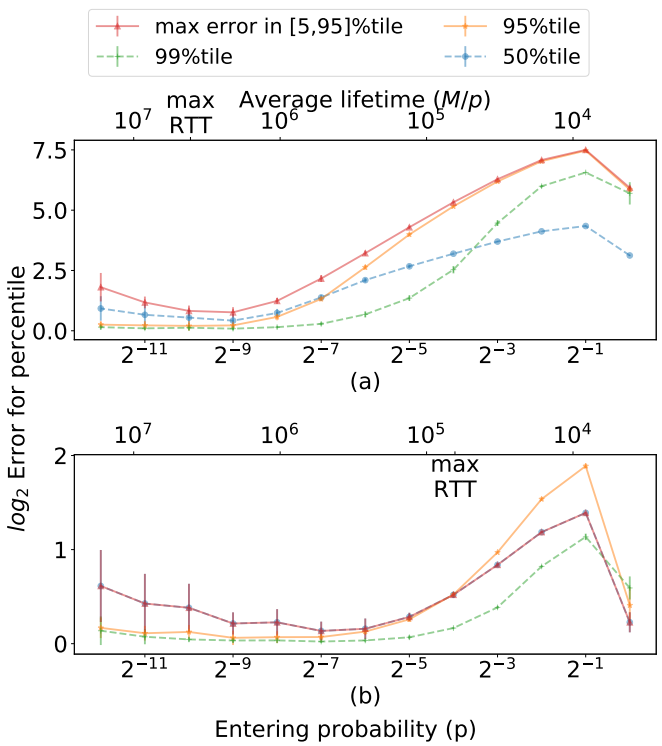

Figure 3: For a fridge with $M = 2^{12}$ memory size processing synthetic trace with ground truth delay between $(0, 2^{12})$ms (top), error is minimized around entering probability $p = 2^{-9}$; for ground truth delay in $(0, 2^{6})$ms (bottom), the best p increases to around 2^{-4}, which leads to a shorter average lifetime in the fridge.

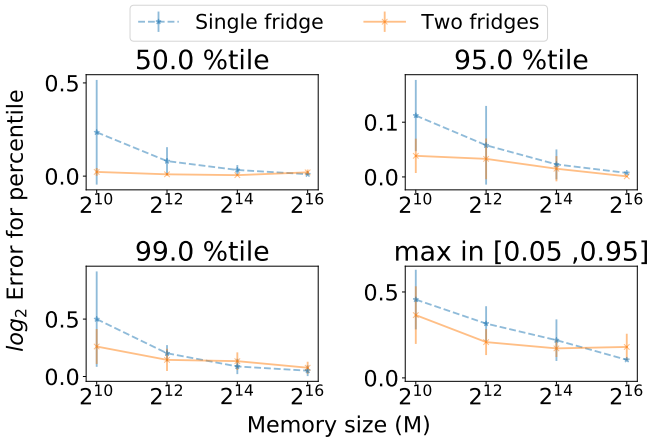

Figure 4: Compared with the single fridge single stage variant, using two fridges provides more benefits as memory size M decreases.

number of insertions between request-response pairs that experience the maximum delays we are interested in measuring. In Figure 3, we show the error of the fridge algorithm under different entering probabilities, under a small memory size $M = 2^{12}$. We use two different synthetic traces, one with delay range $(0, 2^{12})$ms and another with delay range $(0, 2^{6})$ms.

In Figure 3(a), the largest delay 2^{12}ms corresponds to $4{\times}10^{6}$ insertions between request and response. The best choices of p indeed appear near average lifetime $M/p = 4 \times 10^{6}$. For Figure 3(b), the largest delay 2^{6}ms corresponds to 6.4×10^{3} insertions, and we observe an increase in the best choice of p (thus decreased average lifetime). Also, the fridge's accuracy is not very sensitive to the exact choice of p, as we observe similar accuracy when choosing any p within 0.5x-2x of the optimal.

We conclude that network administrators deploying the algorithm should provision p according to the expected maximum delay to be measured in the network, by aligning the fridge's average lifetime to the number of insertions under this delay. It is likely not necessary to tune p continuously to adapt to the slight temporal changes in traffic pattern, as the fridge is robust against a 0.5x-2x change in optimal p, and we only observe a 1.2x-1.3x diurnal change in per-minute average traffic rate in our network. However, p should be re-calibrated whenever the traffic rate or the average delay changed more than 2x.

5.3 Beyond a single fridge

We now show that using multiple fridges improves accuracy by experimenting with a two-fridge algorithm.

The two fridges each use half of the total memory size ($M_1 = M_2 = \frac{M}{2}$), and we find the best entry probabilities (p_1, p_2) using grid search; the outputs of two fridges are then combined using Inversed Variance Weighting (as discussed in § 4.2) to produce the final estimated CDF. In Figure 4, we show the two-fridge algorithm is more accurate than a single fridge when processing the real-world traffic trace, especially in the more challenging regime with smaller memory size.

Finally, we compare our single- and two-fridge algorithms with both the single-stage and the four-stage naive algorithms. We similarly tuned the naive algorithm to use the best expiration threshold for each memory size. Figure 5 shows that the two-fridge algorithm performs consistently better than the naive algorithms, achieving the same accuracy using 2x-4x smaller memory.

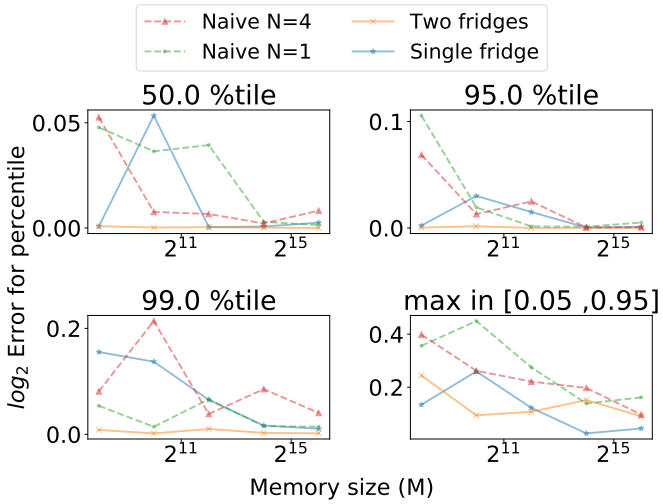

Figure 5: The single- and two-fridge algorithms exhibit much lower estimation error for 50, 95, 99th-percentile delays, compared with the 1- and 4-stage naive algorithms, under various total memory sizes.

It is not yet clear how to systematically find the best entry probability for more than two fridges without using exhaustive search, or how much further we can reduce error by using three or more fridges. We leave these as future work.

6 Hardware Implementation

We build a prototype of the fridge data structure that runs on the Barefoot Tofino high-speed programmable switch, that can measure traffic delay distribution at line-rate of 100 Gbps per port across 16 ports (1.6 Tbps aggregated throughput). The prototype program is written in the P4 [22] language and has approximately 800 lines of code. We plan to make the source code of our prototype publicly available with the final version of this paper.

Our prototype measures TCP handshake delay distribution by parsing handshake packets, hashing the IP address pair, port number pair, and TCP sequence number together into a request/response ID; it is straightforward to adapt the program to measure other delays by similarly defining and calculating request/response IDs using other information in the packets.

In this section, we briefly discuss some technical details about implementing the algorithm on hardware programmable switches, and present evaluations of our prototype.

6.1 Implementing the fridge table

We implement the fridge using three hash-indexed arrays: $IDS[\cdot]$, $TS[\cdot]$, and $CTR[\cdot]$, storing the request packet's ID, its timestamp, and the insertion counter, respectively.

When a request arrives, we first combine the relevant packet header fields (and compress them through a digesting hash function) to generate a 32-bit request ID, as well as generating a random array index $idx = hash(ID) \in [M]$, which specifies a location in the fridge. Subsequently, we invoke the pseudorandom number generator to generate a 32-bit random number $r \in [0, 2^{32})$, and compare it against a threshold: the request is only inserted if $r < p \cdot 2^{32}$. This way, we implement the entry probability p as the request is ignored with probability $1 - p$. We also maintain an additional register as the fridge's insertion counter, which is incremented by one for each inserted request. Subsequently, we write this request's ID, the current timestamp, and the insertion counter into $IDS[idx]$, $TS[idx]$, and $CTR[idx]$, respectively.

When a response arrives, we calculate the same ID and index $idx = hash(ID)$, and check if a matching ID is currently stored in $IDS[idx]$. If so, we generate a delay sample by calculating the delay (the difference between current timestamp and $TS[idx]$) and the number of survived insertions (x, the difference between current insertion counter and $CTR[idx]$), and also erase the current values. Otherwise, if the stored ID mismatched, the request didn't survive and we simply do not produce a sample.

We note that the process of generating IDs using a 32-bit hash digest might lead to mismatching request and response sharing the same ID. However, the probability for a hash collision when generating an ID is much lower than hash collisions on the shorter idx (8-16 bits), therefore it has negligible impact to producing delay samples. Requests have a much higher chance to be evicted than surviving and seeing a mismatching response with the same ID.

6.2 Correcting the bias

Given the delay t and survived insertion count x in a reported sample, we calculate the single-fridge bias correction factor in the switch data plane.

As the programmable switch only supports basic arithmetic operations, we cannot exactly calculate $p^{-1}\left(1-\frac{p}{M}\right)^{-x}$; instead, we notice p and M are known constants, and exploit P4's match-action semantics to match x with a list of prefixes, effectively building a lookup table with pre-computed x ranges and the corresponding correction factor.

The prefix matching logic available on the programmable switch was originally used for routing network packets over the internet by IP address prefixes. Given that the bias correction factor $p^{-1}\left(1-\frac{p}{M}\right)^{-x}$ is a monotonic function over x, it is straightforward to implement a lookup table that

matches on the bit prefixes of the binary representation of x and outputs the correction factor. To save memory, we do not implement all possible correction factors exactly, and instead only maps x approximately to several integer correction factors starting from $1/p$ with 1.1x increment. The programmable switch hardware supports matching using different bit prefix lengths, which is very handy given that the correction factor has x in its exponent.

For example, with $p = 0.5$ and $M = 2^{16}$, the first prefix-matching rule matches $x \in [0, 7\times2^{11})$ and outputs correction factor 2, and the next rule matches $x \in [7 \times 2^{11}, 9 \times 2^{12})$ and outputs correction factor 3. We use a python script to automatically generate these rules based on p and M.

Subsequently, we tally the delay samples and build a histogram in a register array, by adding up the correction factors of delay samples that fall into certain delay ranges. In our prototype, we maintain a histogram with 32 bins using $log(t)$ as the bin index, however it is straightforward to discretize the distribution differently or use more bins.

This implementation allows the data-plane program to track the distribution of delay in real time, enabling diverse applications such as real-time SLA monitoring and dynamic rerouting. We can either maintain an overall delay distribution, or split the traffic into different subsets and maintain a separate delay distribution for each subset.

Still, we note that various approximations incur additional error in the measurement. One may collect all the produced samples and perform the bias correction outside of the data plane, using a program running on a server (with no arithmetic constraints), to exactly calculate the correction factors and the delay distribution. This also allows running the more complex correction operations required by the multi-fridge algorithm.

6.3 Prototype evaluation

We evaluate our prototype fridge implementation by running it on a Barefoot Tofino Wedge-32X programmable switch, processing the same real-world traffic trace used in § 5. We use the MoonGen [11] traffic generator to replay the trace to the switch at real-world speed, by reading the pcap file from a ramdisk. The traffic generator runs on a server with two 10-core Intel Xeon 4114 CPUs and a Mellanox ConnectX-5 100 Gbps NIC, using Ubuntu 20.04 and DPDK 19.05.

We check that the fridge is producing samples correctly, by running it under various memory size M and collecting all the samples reported using a server running packet capturing. Analyzing the raw samples produce a delay distribution CDF closely matching the ground truth, unless M is set to be very small. The results closely match what we observe under simulation. We also analyze the effect of approximating the correction factor using a lookup table, and find it only negligibly affects the resulting CDF: we observe a maximum relative error between 0.2% and 0.9%, about ten times smaller than the relative error between the fridge's estimated distribution and the ground truth.

A fridge with $M = 2^{16}$ entries costs about 6.4% of the total register memory available on the programmable switch. We only need $M = 2^{12}$ entries to process the real-world traffic trace used in § 5 and produce an accurate delay CDF, and under this configuration we only consume 1.7% of the total register memory. Besides the register memory allocated for the fridge, the prototype program also uses 23.6% of hash units (for array indexing), 7.3% of Ternary Content Addressable Memory (TCAM, for prefix lookup tables) and less than 5%-10% of any other hardware resource.

Given that we only use moderate hardware resource, we believe our prototype program's performance is sufficient to process traffic at the switch's maximum line rate; unfortunately, our packet generator server can only replay trace at speeds up to 8 Gbps (due to single-core CPU bottleneck) and generate synthetic traffic at approximately 80 Gbps. We have validated the prototype behaved correctly under both cases. At 80 Gbps, the prototype data-plane program is processing more than 160 million requests and responses per second, which is 80 times faster than a simulator written in C++ (processing 2 million requests/responses per second on a single CPU core).

7 Related Work

Measuring delay. PingMesh [14] and NetBouncer [21] measure round-trip delay by running active measurement on end hosts, and calculating the time difference between outgoing probes and incoming replies. Active measurement can generate comprehensive reports periodically, however the probe might not experience the same delay as actual application traffic [2]. Meanwhile, Ruru [10] and Abut [1] measure TCP handshake delay by passively observing the three-way handshake packets. This is helpful for producing a flow-level distribution of delays. Veal et al. [23] measure delay for all TCP packets, by adding a timestamp as a TCP option header. This method can produce accurate samples, as long as intermediate firewalls do not drop the option header and the client correctly echoes back the timestamp. Instead, Jiang and Dovrolis [16] passively measured TCP packets by observing sequence numbers, and produce delay estimates for many but not all packets. However, these methods all require exporting a large number of packets from the data plane for off-path analysis, which incurs significant networking and computational overhead when measuring high-speed networks.

Delay measurement in the data plane. Dapper [13] and Chen et al. [6] both measure delay directly in the switch

data plane. Dapper tracks TCP flows individually, and produces one delay sample per round-trip for each TCP flow; this requires pre-allocating memory for every TCP flow being tracked. Meanwhile, [6] works directly with packets from all flows. This allows better memory utilization (almost the entire memory is used at all times) and does not require per-flow state, however it leads to the bias issue against long delays, which we addressed in this work.

Quantile sketch. KLL [17] and DDSketch [19] are quantile sketches that approximately measure samples in a distribution and produce an estimate of certain quantiles using only small memory. QPipe [15] implements a quantile sketch that runs fully within the programmable switch data plane. Our work does not measure quantiles directly as we only re-weight the produced samples to ensure the resulting distribution is unbiased, and we rely on subsequent post-processing to aggregate the samples and produce statistics such as quantiles. It is possible to feed the samples and their weights output by a fridge into a quantile sketch, such that we can approximately answer queries about percentile delay without the need to save the entire distribution.

8 Conclusion

In this paper, we show how to compute unbiased estimates of delay in the data plane, using the *fridge* data structure that tracks the number of evictions while the request remains in the fridge. By correcting for the probability of eviction due to hash collisions, we can produce accurate delay distributions that closely match the ground truth. Evaluation shows that our algorithm is indeed much more accurate at estimating delay percentiles, compared with prior works using the same amount memory. The two-fridge algorithm achieved the same accuracy while saving 2x-4x memory. We also build and validate a prototype implementation of the fridge running on high-speed programmable switches, that measures the unbiased delay distribution accurately and efficiently within the data plane.

There are still many network performance metrics we struggle to measure in the data plane due to the limited memory size. The fridge design opened up many possibilities measuring unbiased statistics for sophisticated "join-over-time" queries, where two or more packets across the traffic stream need to be joined together. One question remain unsolved by this paper is whether our fridge design is applicable to statistics beyond the distribution of delays between request-response pairs, allowing us to estimate distributions of other more complex metrics such as the total number of bytes or packets in a completed flow. It is straightforward to accumulate the counts in the fridge and produce a correction factor for each sample, and we can obtain an unbiased estimate for the frequency of each count using the same correction factor. However, further empirical analysis are needed when we start to aggregate these counts into more complex metrics, such as their quantiles and skewness, as the unbiased property cannot be carried over automatically. We are also excited to apply the idea of correcting survivorship bias in other randomized data structures in our future work.

References

[1] Fatih Abut. 2019. A distributed measurement architecture for inferring TCP round-trip times through passive measurements. *Turkish Journal of Electrical Engineering & Computer Sciences* 27, 3 (2019), 2106–2120.

[2] K Auerbach. 2004. Limitations of ICMP Echo for network measurement. https://iwl.com/idocs/limitations-of-icmp-echo-for-network-measurement.

[3] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *IEEE ICNP*. 313–323.

[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM Conference*. 99–110.

[5] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*. Springer, 693–703.

[6] Xiaoqi Chen, Hyojoon Kim, Javed M Aman, Willie Chang, Mack Lee, and Jennifer Rexford. 2020. Measuring TCP round-trip time in the data plane. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure*. 35–41.

[7] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. dRMT: Disaggregated programmable switching. In *ACM SIGCOMM Conference*. 1–14.

[8] William G Cochran. 1954. The combination of estimates from different experiments. *Biometrics* 10, 1 (1954), 101–129.

[9] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: The Count-Min Sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[10] Richard Cziva, Christopher Lorier, and Dimitrios P Pezaros. 2017. Ruru: High-speed, Flow-level Latency Measurement and Visualization of Live Internet Traffic. In *ACM SIGCOMM Posters and Demos*. 46–47.

[11] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*. Tokyo, Japan.

[12] Cristian Estan and George Varghese. 2002. New directions in traffic measurement and accounting. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*. 323–336.

[13] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. 2017. Dapper: Data plane performance diagnosis of TCP. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*. ACM, 61–74.

[14] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. 2015. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM*, Vol. 45. ACM, 139–152.

[15] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. 2019. Qpipe: Quantiles sketch fully in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And*

Technologies (CoNEXT). 285–291.

[16] Hao Jiang and Constantinos Dovrolis. 2002. Passive estimation of TCP round-trip times. *ACM SIGCOMM Computer Communication Review* 32, 3 (2002), 75–88.

[17] Zohar Karnin, Kevin Lang, and Edo Liberty. 2016. Optimal quantile approximation in streams. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 71–78.

[18] Zaoxing Liu, Samson Zhou, Ori Rottenstreich, Vladimir Braverman, and Jennifer Rexford. 2020. Memory-efficient performance monitoring on programmable switches with lean algorithms. In *Symposium on Algorithmic Principles of Computer Systems*. SIAM, 31–44.

[19] Charles Masson, Jee E Rim, and Homin K Lee. 2019. DDSketch: a fast and fully-mergeable quantile sketch with relative-error guarantees. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2195–2205.

[20] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *ACM SIGCOMM Symposium on SDN Research*. 164–176.

[21] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. 2019. NetBouncer: Active Device and Link Failure Localization in Data Center Networks. In *USENIX Networked Systems Design and Implementation*. 599–614.

[22] The P4 Language Consortium. 2018. $P4_{16}$ Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html.

[23] Bryan Veal, Kang Li, and David Lowenthal. 2005. New methods for passive estimation of TCP round-trip times. In *International Workshop on Passive and Active Network Measurement*. Springer, 121–134.