Purdue University
Purdue e-Pubs

Computer Science Technical Reports | Department of Computer Science

2011

Leave Them Microseconds Alone: Scalable Architecture for Maintaining Packet Latency Measurements

Myungjin Lee
Purdue University, mjlee@purdue.edu

Nick Duffield
AT&T Labs-Research

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

Report Number:
11-013

Lee, Myungjin; Duffield, Nick; and Kompella, Ramana Rao, "Leave Them Microseconds Alone: Scalable Architecture for Maintaining Packet Latency Measurements" (2011). *Computer Science Technical Reports.* Paper 1744.
http://docs.lib.purdue.edu/cstech/1744

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Leave Them Microseconds Alone: Scalable Architecture for Maintaining Packet Latency Measurements

Myungjin Lee[†], Nick Duffield[‡], Ramana Rao Kompella[†]
[†]Purdue University, [‡]AT&T Labs–Research

ABSTRACT

Latency has become an important metric for network monitoring since the emergence of new latency-sensitive applications (*e.g.*, algorithmic trading and high-performance computing). To satisfy the need, researchers have proposed new architectures such as LDA and RLI that can provide fine-grained latency measurements. However, these architectures are *fundamentally ossified* in their design as they are designed to provide only a specific pre-configured aggregate measurement—either average latency across all packets (LDA) or per-flow latency measurements (RLI). Network operators, however, need latency measurements at both *finer* (*e.g.*, packet) as well as *flexible* (*e.g.*, flow subsets) levels of granularity. To bridge this gap, we propose an architecture called MAPLE that essentially stores packet-level latencies in routers and allows network operators to query the latency of arbitrary traffic sub-populations. MAPLE is built using scalable data structures with small storage needs (uses only 12.8 bits/pkt), and uses optimizations such as range queries to reduce the query bandwidth significantly (by a factor of 10 compared to the naive).

1. INTRODUCTION

For the longest time, networking engineers and researchers have focused mainly on obtaining high end-to-end throughput in IP networks. In recent years, however, *latency* has evolved into a metric that is as important as throughput in IP networks. While low latency is a desirable property for any network-based application, this obsession towards low end-to-end latency stems from the stringent requirements of many new kinds of datacenter, cloud, and wide-area applications that have become popular in the recent times. For instance, several cloud applications (*e.g.*, Salesforce, Google App Engine, modern Web services) involve complex back-end processing, such as accessing storage, SQL database transactions, etc. After removing time for computation and wide-area RTTs, the budget for datacenter network accesses is significantly reduced. Similar requirement exists for partition/aggregate type workloads found in search, social collaboration applications, where jobs that do not finish within a certain time are typically cancelled thus affecting the overall result, and in some cases, lost revenue [5].

Even more stringent latency requirements, in the order of 10s of μseconds, exist for high-performance computing (HPC) applications [1] and financial trading applications [28]. Finally, although in the past Internet applications did not require as stringent requirements, with the advent of multimedia applications and social gaming, latency requirements have become more stringent. Many ISPs today provide iron-clad SLAs that leave little room for latency spikes outside of the propagation delays [3].

Network operators managing such latency-sensitive applications need sophisticated tools for high-fidelity latency measurements at various places in the network that will help them identify root causes of SLA violations, determine offending applications that may hurt the performance of others, perform traffic engineering and so on. In light of the importance of these measurements, there has been some recent research on developing measurement mechanisms such as the lossy difference aggregator (LDA) [23] and reference latency interpolation (RLI) [24].

A *key limitation* of these existing techniques is that they only obtain latency measurements at the granularity of a fixed pre-configured aggregate (across all packets in LDA, and per-flow latencies in RLI). By making the granularity of the aggregates for latency measurements part of the architecture, these prior architectures are quite *ossified*, lacking flexibility to obtain arbitrary latency measurements than what they are already pre-programmed to achieve. What network operators need instead is a holistic architecture that provides the ability to obtain arbitrary latency measurements from switches. Such an architecture would help network operators with powerful tools to help debug and manage low-latency applications in their networks. Designing such an architecture is the main objective of this paper.

In our quest to obtain arbitrary latency measurements from switches, we ask ourselves, "What is the finest granularity of latency measurements a network operator may be interested in ?" The LDA and RLI architectures implicitly assumed that aggregate or flow-level granularity of measurements is what operators may care about. In this paper, we argue instead that there are many compelling scenarios where finer-granularity measurements may be important. For example, for diagnosing client delays in online services, it may be

critical to know whether a DNS query (that is typically a single packet) got delayed, or whether a backend transaction got delayed in the network, or whether there were processing delays. Similarly, for financial trading applications, one may care about the delay of a single stock trade (that may be carried in one packet). For HPC applications built on message passing libraries (*e.g.*, MPI), latencies of even single messages may be quite important. In addition, one may wish to focus on latencies for a *subset* of packets that belong to a flow, perhaps to hone in on the ones that exhibited abnormal latency, or to track the latency time-series of the flow. Thus, clearly, in order to satisfy these requirements, we need a more flexible architecture than the one-size-fits-all approach of existing solutions such as LDA (aggregate) or RLI (per-flow).

Since the finest granularity of latency measurements is on a *per-packet basis*, we start with an architecture that achieves these measurements in a scalable fashion. Then, any other forms of aggregation (per-flow, per-prefix, all packets), that may be of importance to network operators, are easily composable from these packet-level measurements. Such an architecture essentially decouples the collection of measurements (at the granularity of a packet), and aggregation (across arbitrary subpopulations) during query time. This key intuition forms the the basis for our proposed architecture MAPLE.

MAPLE essentially consists of two main components—a scalable packet latency store (PLS) and a query engine—at each router. PLS stores the latencies of all packets that appear at the router. In high speed networks, storing all the packets and their associated latencies is going to be expensive; hence we store latencies only for a small amount of time (*e.g.*, 1s) in high-speed SRAM, and rely on flushing them periodically to a higher capacity data store. Since storing the entire packet and delay requires high storage (in terms of bits/packet), we propose a novel approach that first clusters packets and associates a delay value for each cluster, and then, uses a novel hardware data structure called shared-vector Bloom filter (SVBF) to significantly reduce the memory requirement. SVBF makes the architecture technologically feasible in high-speed switches where SRAM is a very scarce and precious resource. We show how the entire PLS can enable an efficient streaming implementation in hardware that can keep up with line rates.

The second component of MAPLE, the query engine, essentially allows end-hosts or a centralized entity to initiate a query for the packet. These queries need to be within a particular timeframe of the original packet, otherwise, the store in the router may not have any history of the packet. By constructing queries across arbitrary packets, end-hosts can easily obtain per-packet latencies for all (or a sample) of packets within a given subpopulation, using which they can compose the aggregate latency measurements for that flow. We also consider mechanisms to reduce query bandwidth by providing the ability to perform range queries.

Thus, the main contributions in this paper are as follows:

1) We propose MAPLE for maintaining per-packet latency measurements in a scalable fashion. Our architecture allows network operators to obtain any aggregate of measurements thus subsuming the functionality of existing architectures, while providing newer and more powerful capabilities such as computing aggregate latency measurements across any sub-populations.
2) We propose novel mechanisms that use streaming algorithms for clustering packet delays, and storing them compactly using a novel data structure called SVBF that is much more storage efficient (requires only 12.8 bits/pkt) than a varaint of regular hash tables (that may require 147 bits/pkt) and also minimizes memory accesses for inserts and lookups. We also propose range queries to reduce the amount of query bandwidth required.
3) We built a software prototype of our architecture. In our evaluations, we found that MAPLE achieves lower per-packet latency estimate error (almost 6× lower) compared to prior data structures for comparable storage. We also found that the range query achieves significant reduction in query bandwidth (almost 10×) compared to packet query.

The rest of the paper is organized as follows. We present our measurement goals and the high-level ideas of our architecture in §2. We discuss the two major components of our architecture—packet latency store and query engine in §3 and §4 respectively. We then discuss our evaluation of the architecture in §5. We briefly outline how we can implement our architecture in high-speed routers in §6, followed by related work in §7.

2. MAPLE ARCHITECTURE

In this section, we outline a flexible architecture for obtaining high-fidelity latency measurements in the network. Before we describe the architecture, we first state our measurement goals followed by a brief discussion on why previous solutions cannot satisfy these goals.

2.1 Measurement goals

Our goal is to enable a high-fidelity latency measurement architecture that satisfies the following requirements:

- *R1) Per-packet latency measurements.* The architecture should allow operators to obtain latency information about a single packet at various routers in the network.
- *R2) Measurements across arbitrary aggregates.* It should enable network operators to compute measurements across arbitrary aggregates (*e.g.*, per-prefix, application, flows, sub-flows) easily.
- *R3) Measurements across arbitrary locations.* We need support for latency measurements both within and across routers to allow network operators complete freedom to selectively turn on interfaces between which they need measurements.

Such an architecture will provide detailed latency infor-

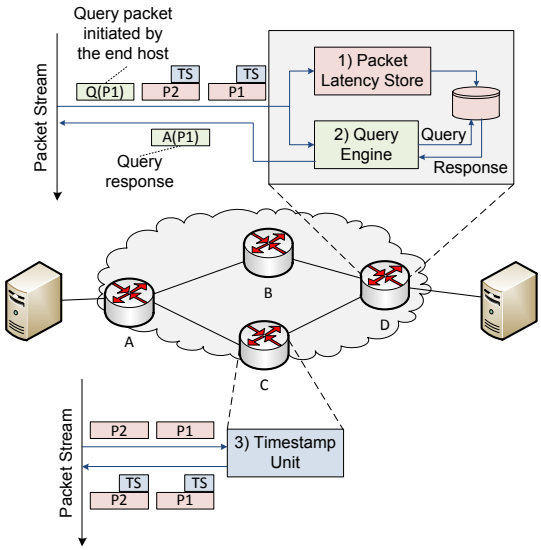


Figure 1: MAPLE architecture.

mation that will help network operators to debug their networks and satisfy the demands of modern latency-sensitive network applications.

2.2 Limitations of previous solutions

We consider mainly passive measurement solutions since, as pointed by prior work [23, 24], it is difficult to estimate packet-level latencies by injecting active probes. As mentioned before, there have been mainly two solutions, LDA [23] and RLI [24], that have been proposed for fine-grained latency measurements in the recent past. (Another solution for per-flow latency measurements is Consistent NetFlow [25], but it is quite similar in spirit to RLI, and arguments applicable to RLI are applicable there too.)

LDA provides only aggregate measurements and cannot be modified simply to satisfy all the requirements listed before. For instance, it cannot satisfy $R1$. Although $R2$ could potentially be satisfied by configuring different LDAs on a per-aggregate basis, but this approach will not scale well, as all potential aggregates need to be pre-configured, and there can be many such aggregates. RLI provides per-flow measurements, and hence, it can satisfy $R2$ partially for aggregates that can be obtained from individual per-flow measurements. For example, one can aggregate all the flows per-prefix from per-flow measurements but *cannot* obtain finer granularity measurements than flow, such as a packet (or flow subsets). In addition, LDA and RLI cannot easily satisfy requirement $R3$, because of the FIFO ordering assumption in LDA and temporal delay correlation assumption in RLI, both of which may not hold true across arbitrary measurement points.

2.3 Architecture

Given existing solutions fall short of satisfying the requirements we outlined earlier, we propose Measurement Architecture for Packet LatEncies (MAPLE). Our architecture (shown in Figure 1) is based on three key ideas: First, in order to satisfy requirement $R1$, we need MAPLE to store per-packet latency measurements in some scalable way; any form of aggregation within routers cannot be used to satisfy $R1$. Thus, at its heart, MAPLE contains a scalable packet latency store (called PLS) designed to simply store latencies of *all* packets in a scalable and efficient fashion. Since storing all packets will mean significant storage requirement that will be prohibitively expensive if not technologically infeasible, it only stores packet latencies only for a small amount of time, τ, say 1-100s, in high speed memory. Every τ seconds, the store will be (optionally) flushed to an off-chip secondary storage (*e.g.*, DRAM, SSD), where it can be held for a longer time. We discuss PLS in §3.

Second, to satisfy requirement $R2$, it essentially contains a generic query engine that allows clients to query switches about the latency of a packet (using its hash, for instance) that has traversed that particular switch. These queries need to be within the storage timeframe (either high-speed or optional off-chip storage), otherwise the switch may lose the packet latency record. We expect that queries are mostly going to be only for specific applications or customers which are experiencing trouble. Instead of a per-packet query, we also provide range query mechanisms (more details in §4) to reduce query bandwidth. By querying latencies of specific packets that form an aggregate, the client can obtain latency measurements across arbitrary sub-populations. We discuss the query component in more details in §4.

Finally, it is simpler to satisfy requirement $R3$, if we enable packet headers to carry timestamps (as shown in Figure 1). While we understand that it may be complicated in the short-term to make header changes (as prior work [24, 23] pointed out), we believe this is a cleaner longer-term option that switch vendors are already considering. In our discussions with prominent switch vendors, they mention that implementing a timestamp within the switch is not a problem, and, in some cases, such as Fulcrum [2], they already have an internal timestamp within the switch. Thus, the assumptions in LDA (FIFO ordering) and RLI (temporal delay correlation) are no longer required in our architecture thus enabling a more flexible architecture. However, if timestamping is not feasible, for restricted settings, such as packets within a router, or when there is a series of routers in a FIFO order, we can still use temporal delay correlation assumption made in prior work [24] and obtain approximate delays on a per-packet basis. In that sense, our architecture builds on *any* scheme to obtain one-way delay for each packet.

Note that in all cases, we assume high precision time synchronization (similar to prior solutions LDA and RLI) between the two measurement points, which has become feasible in modern times due to the increasing adoption of IEEE 1588 [16] and GPS-enabled clocks.

3. PACKET LATENCY STORE (PLS)

Simply put, the goal of PLS is to store a packet and its associated latency value in a scalable fashion. In normal settings, this goal would be relatively straightforward to accomplish using a simple linear-probing or linked-list-based hashtable implementation [13]. Unfortunately, hashtables are not efficient in their storage since they typically require storing a packet hash (32 bits) and associated timestamp (20 bits). For a million packets, we need about 50 Mb which is already quite expensive; for an OC-192 link, even this storage will last only 0.2s unfortunately (assuming 5M packets per second). Even if we use a slightly coarser timestamp (say 10 bits), it will reduce the memory need only slightly. Besides, linear-probing or linked-list-based hashtables are not easy to implement in a hardware in a pipelined fashion, and may incur unpredictable insert times (depending on the number of accesses required to find an empty slot, or the length of the collision chain).

If we need to store *precise* latency values for each and every packet, relying on hashtables is probably the best recourse unfortunately. Luckily, for the kind of applications we envision, such as performance diagnosis or detecting SLA violations, we can exploit the fact that *the latency values for each and every packet need not be precise, and can be approximate instead*. If we assume some amount of inaccuracy is tolerable, then we can significantly reduce the memory usage—this is the key intuition behind our approach.

Our approach In our approach, we exploit two key ideas. First, within a given measurement interval, there are typically only a few dominant latency values (depending on the utilization) where most of the packet values are clustered. In the worst case, the latency values can be all over the entire permissible range, but in general, this is typically rare. Thus, instead of storing packets and their associated timestamps, we can first cluster packets into *equivalence classes* based on the delay values, and associated a single delay value, called *cluster center*, for all packets within the cluster. Second, for each cluster, we can leverage approximate membership query data structures such as Bloom filters [8], that have gained significant prominence in networking applications recently, for better efficiency in storage (in terms of bits/packet) as well as implementation in hardware (just a bit vector and few hash functions). We discuss these in more detail next.

3.1 Selecting representative delays

Depending on whether the clusters are chosen statically or dynamically, there are two broad choices for selecting the cluster centers. For the static case, we consider logarithmic center selection, while we explore online clustering algorithms (k-means and k-medians) for determining centers dynamically.

Logarithmic delay selection. In this method, we first select a range of latency values that packets can experience, and then divide this range into logarithmic sub-ranges. For instance, if the delay range is 0.1-10,000 μs, we have 5 sub-ranges; 0.1-10 μs, 1-10 μs, and so on. If we have n sub-ranges, we assign k/n representative delays linearly for each sub-range. If k = 50, and n = 5, each sub-range assigns 10 representative delays linearly. While this method does not take the pattern of delays into account, the complexity of choosing representative delays is minimal. Because the distance between two center delays in a sub-range is equal, the relative and absolute error of a packet latency estimate remains bounded and stable regardless of packet delay distributions. However, accuracy may not be close to the optimal accuracy as we can obtain with given k delay centers.

k-means and k-medians clustering. If we know distribution of packet delays in an interval *a priori*, selecting representative delays can be formulated as a clustering problem. In literature, there are two broad classes of algorithms—k-means and k-medians—that can help determine good cluster centers [20]. Typically, both types of algorithms minimize the average absolute error of packet latencies, because they choose centers that minimize total sum of distances between each member with its closest center.

Formally, for observations $x_1, x_2, \ldots, x_n$, the k-means algorithm aims to partition them into k sets, $S_1, \ldots, S_k$, $k \leq n$, so as to minimize the sum of squares of distances within cluster from the center (mean), i.e.,

$$\arg\min_{\{\mu_i\}} \sum_{i=1}^{k} \sum_{x_j \in S_i} ||x_j - \mu_i||^2$$

where μ_i is the mean of the cluster S_i. k-medians clustering algorithms minimize the distance to the median of a cluster as opposed to the least-squares distance that k-means obtains. The advantage of k-medians is that it is more resilient to outliers which have too large or small values.

There are two key concerns with using these algorithms directly in our setting though. First, the basic algorithms cannot be directly implemented in a streaming fashion due to their high run-time complexity, $O(n^{k+1} \log n)$. There exist heuristics such as the classic Lloyd's algorithm [27], but still it can be quite computationally intensive. Second, the centers are determined *after* running the algorithm on all the packets in a given measurement interval, but we need the centers to be determined *before* the packets start streaming in. We discuss how to address these issues next.

3.2 Streaming clustering

In order to address the first problem concerning the high run-time complexity, we use a streaming version of k-medians clustering algorithm complexity in our architecture. For the second problem of lagged availability of centers, we use a pipelined architecture, where computed centers from a previous epoch are used to cluster packets for this epoch.

Online version of clustering algorithm. There exist a few time-efficient k-medians clustering algorithms [10, 19, 18] in literature. In our architecture, we leverage an online clustering algorithm proposed in [10] because the algorithm makes

no assumption about the characteristics of the streaming data and is space-efficient. We need to make several modifications, which we describe later, to make the algorithm more efficient than the version in [10]. We describe briefly how the algorithm works and our modifications to speed up the algorithm next.

Given a stream of n data points and k centers we wish to find, this algorithm consists of two stages—online and offline clustering. At a high level, the online clustering stage works in many (not necessarily equal) phases over the entire epoch to find $O(k \log n)$ candidate medians that the offline clustering stage subsequently reduces to k centers. In each phase, it uses Meyerson's online facility location algorithm [29], and chooses to open a new center with probability δ/f, where δ is the distance of the current point x to the closest already-open center, and f is the cost. In this algorithm, cost f is $L/(k(1+\log n)$, where L is the lower bound cost of the optimal. Note that L is refined at the beginning of every phase by multiplying the previous value of L by a fixed constant β (we use $\beta = 34$ in our implementation). (Refer to the PolyLogarithmic Space algorithm in page 4 of [10] for the exact description.) The current phase is terminated if either the number of opened centers or the associated cost function exceeds some threshold (details in [10]). The algorithm terminates when all the packets are consumed, and leaves behind a set of $O(k \log n)$ candidate medians.

There are a few modifications we make to the original algorithm to contain the run-time complexity. First, since speed is critical, we use only one thread instead of $2 \log n$ parallel threads in the original algorithm in [10] (see PARACLUSTER in page 4 of the paper). Second, the online algorithm requires searching for the closest existing center out of $O(k \log n)$ centers in each phase, which is hard even for small k to do in 1 cycle. We therefore run on the online algorithm only on *sampled* packets; a 10% sampling rate trivially gives 10 cycles to do these lookups. For $k = 50$ and $n = 400,000$, we observe about 1000 centers which can be looked up with a balanced binary tree using 10 memory accesses. We observe in our evaluations that 1-10% sampling rate has virtually no effect on the quality of the centers produced by the algorithm.

Handling lagged availability of centers The problem here is that we cannot compute the centers and cluster on the same packet stream in one pass. Besides, the streaming algorithm itself operates in two stages, online and offline. To address this problem, we design a three-stage pipeline consisting of the following stages to handle this issue: The first stage consists of the online clustering algorithm that computes the $O(k \log n)$ centers that operates on packets in epoch i. The second stage consists of the offline clustering which will result in k centers by consuming these $O(k \log n)$ centers. Finally, we cluster the packets in epoch $i + 2$ depending on the closest center that matches the packet's latency in the final stage. Since these stages operate in a pipelined fashion, the centers computed will be based on the dynamics of

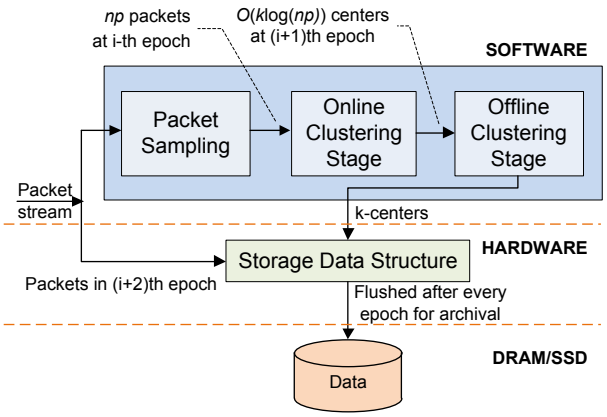


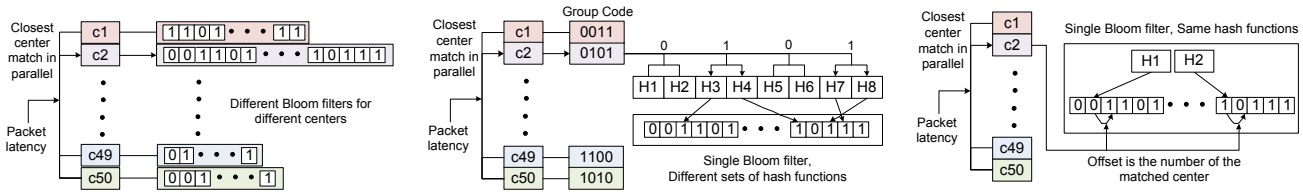
Figure 2: Architecture for streaming representative delay selection and storage provisioning.

packets computed *two epochs back*. Assuming some amount of stationarity across measurement intervals (our evaluation shows holds true in practice), this pipelined approach should work well. However, to cover for the worst case where these centers may be significantly dissimilar to each other, we propose a hybrid clustering approach that combines static allocation using the logarithmic centers with the dynamic allocation approach using the pipelined implementation of k-medians algorithm as follows.

Hybrid clustering The basic idea of hybrid clustering is to choose $k/2$ centers with logarithmic delay selection method and rest $k/2$ centers are computed by streaming k-medians algorithm. To enable hybrid clustering algorithm, we make two more modifications in the online version of clustering algorithm explained earlier. First, at the online clustering stage, the centers chosen by logarithmic delay selection method are always selected as a new center in each phase and the number of data points added to the centers is incremented by one. Second, when $O(k \log(np))$ candidate centers need to be processed at the offline clustering stage, we exclude the $k/2$ centers picked by logarithmic delay selection method from $O(k \log(np))$ centers, the rest candidate centers are fed into the offline algorithm, and finally $k/2$ centers are obtained. While we choose to split the total number of centers equally between static and dynamic allocation schemes, this equal split is somewhat arbitrary and other variants (*e.g.*, 2/3-1/3 split) could also work equally well (although we have not explored this thoroughly yet).

3.3 Storage provisioning

So far, we have reduced the problem of storing $< p_i, l_i >$ tuple to $< p_i, c_i >$ where l_i is the actual latency of packet p_i and c_i is the ith center. Once the k representative delays are selected by the clustering algorithm, we need to determine how much storage is required to store packet latencies. Depending on the data structure that one uses, the actual required memory size can be different. Note that the goal of the data structures is essentially to store and lookup the cen-



(a) PBF, $O(1)$ insert, $O(k)$ lookup (b) COMB, $O(1)$ insert, $O(f)$ lookup (c) SVBF, $O(1)$ insert, $O(1)$ lookup

Figure 3: Different variants of Bloom filters with different insert and lookup and $O(f)$ lookup where f is the size of bit vector. SVBF has $O(1)$ insert and $O(1)$ lookup assuming reading the k vector takes 1 memory access.

ter id corresponding to a packet; the actual latency value corresponding to the center will need to be looked up in a separate table. During the lookup phase, instead of returning the static latency value corresponding to the center, we can dynamically return back the *actual mean* of all the packets that map to a given center. Implementing this would require essentially two additional counters per center (latency sum and packet counts). We call these *refined latency estimates.*

3.3.1 Naive approach: PBF

Given these k cluster centers, we can now simply match each incoming packet latency value to determine the right center, and for each center maintain a separate Bloom filter (BF) in which we record the packet's presence. This naive and intuitive data structure called Partitioned Bloom Filter (PBF) as shown in Figure 3(a).

Insert To store a packet and its latency (l_{pkt}), it finds the right BF corresponding to the latency value by performing a closest center match in parallel (shown in Figure 3(a)). Since the number of centers k is quite small, doing this in parallel in hardware should be relatively easy to do. It then accesses the BF corresponding to this center, and inserts it into the BF just like a regular BF insert, i.e., by hashing using multiple hash functions and setting the bits indexed by the hash values to 1. Since BF inserts are $O(1)$, PBF insert time is $O(1)$, ignoring the small hardware cost for parallel match of the packet latency with the various centers.

Lookup To lookup the latency corresponding to a packet, PBF checks whether the packet is present in any of the BFs; the delay represented by the BF that returns a match is the estimated latency of the queried packet. Clearly, the complexity of the lookup operation is $O(k)$ since k BFs need to be consulted. For a fixed k, since these BFs are strictly partitioned, we can perform these lookups in parallel, thus reducing the complexity to $O(1)$.

Limitations A big problem with PBF is that, PBF needs to allocate storage for each of the k groups differently since the frequency counts for each group could be different. Of course, PBF could use the estimated frequency counts, but since our centers are calculated based on packets two epochs back, these estimates may not be accurate. Thus, the only options are to allocate higher amounts of storage per BF than necessary, or once the capacity of a BF is reached, stop adding packets leading to false negatives.

3.3.2 Prior approach: COMB

The second option we explore is using a recent generic data structure called combinatorial Bloom filters (COMB) designed for supporting multiset membership testing [17]. While PBF strictly partitions all BFs, COMB uses a single BF, but represents different groups using different subsets of hash functions. COMB contains three parameters, an f-bit vector in which θ bits are set to 1 to indicate the code for a group, and h different hash functions for each bit-position (in total $h \cdot f$ hash functions).

Insert For a given group id g_i, it first looks up the code $C(g_i)$ corresponding to the group; for each bit position that is set (θ bits will be set in each code), it picks the corresponding set of hash functions to index into the BF, and set the appropriate bits to 1 (just like a regular BF insert). Thus, for each packet, it requires setting up to $h \cdot \theta$ bits in the BF.

Lookup For each packet, it will test the positions indexed by hashing the packet with all the $h \cdot f$ hash functions. The f-bit code is formed by setting a bit position to 1, only if *all* bits indexed by the hash functions associated with the bit position indicate a 1 in the BF. The code will then correspond to the group id.

Limitations The biggest advantage of COMB is that, unlike PBF, it does not need to know the number of packets in each group. However, storage requirement increases as θ increases, which decreases storage efficiency (as we shall compare shortly). Another limitation is that lookup complexity is high since all the $h \cdot f$ hash functions need to be queried, and all these bits are randomly located.

3.3.3 Our new data structure: SVBF

In the philosophy of maintaining one BF for all the groups, we could consider another, perhaps simpler alternative. Here, instead of storing each packet s_{pkt} directly, we can store its concatenation with the group id, g_i, i.e., $s'_{pkt} = s_{pkt} \oplus g_i$, where $\oplus$ is a concatenation operator. While inserts are fast, lookups (that involve querying a packet with all concatenations of group ids) are quite slow and will take k queries. Further, these cannot be parallelized since all the bits are scattered all across the BF. To address this problem, we propose a new data structure called SVBF that essentially preserves the simplicity of a single BF, but reduces the lookup complexity significantly. Specifically, we store the bits corresponding to different delay values for the same packet close-

Data structure	#Hash	Capacity (m/n)	Insertion	Lookup
hashtable	1	147 bits/pkt	1	1
PBF	9	12.8 bits/pkt	9	450
COMB(50, 1)	9	12.8 bits/pkt	9	450
COMB(11, 2)	7	18.5 bits/pkt	14	77
COMB(8, 3)	6	24.2 bits/pkt	18	48
SVBF	9	12.8 bits/pkt	9	27

Table 1: Example of complexity of storage data structures for single port memory. 32 bit word is assumed for lookup in SVBF. Classification failure rate $p_{\mathbf{CF}} = 0.1$ and $k = 50$. hashtable is tuned for $p_{\mathbf{FC}} = 0.02$.

by so that during queries we can read all the bits in a burst instead of reading them sequentially from various bit positions.

Insert The insert operation is quite similar to a regular BF, except for a small modification. In regular BF, each packet is hashed using multiple hash functions, and bits at those indices are set to 1. In SVBF, we use the hash function index as an offset into a vector of delay values. Thus, we set the bit corresponding to $h_i(s_{pkt}) + g_i$ where $g_i \in [0, k-1]$ is the group number of the packet. This is shown in Figure 3(c) where a packet that matches the second center $c2$ (group id 1) is added into the BF using hash functions H1 and H2. The offset at which the bit is set is 1 for this second center.

Lookup Given a packet s_{pkt}, we first hash the packet to obtain various hash indices $h_i(s_{pkt})$. From each of these bases, we read the next set of k bits, i.e., $h_i(s_{pkt})$ to $h_i(s_{pkt})+k-1$, to obtain bitmaps B_i. We compute the bit-wise AND across all these bitmaps, $B = B_1 \& B_2 \& \ldots$. In the final bitmap (B), the offset where a bit is set to 1 is the group id.

The biggest advantage of this scheme is that, it relies on 'burst reads' which are simpler than random reads that COMB suffers from. Thus, instead of k memory accesses, we only need $\lceil k/w \rceil + 1$ memory accesses for each hash index as shown in Table 2. For example, for $k = 50$, we can obtain the bit maps in a total of $3 \times h$ memory accesses assuming a 32-bit machine word, and h is the number of hash functions. In Table 1, we show an example that outlines the storage complexity, lookup and insertion times of SVBF compared to other data structures.

3.3.4 Classification failures

BFs are typically known to suffer from *false positives* occasionally, in which case a given element may not be in the BF, but the BF may return back with a positive answer. In PBF, this translates to a *classification failure* problem, since two (or more) BFs, one legitimate and one (or more) false positive may both (all) indicate a hit—the question is which one to trust. Similarly, COMB too may suffer from classification failures where more than θ bits in the bit vector are set to one. Finally, even SVBF may suffer from classification failure, since the bit map B described above may have more than 1 position set to 1 occasionally. We formally analyze this in the next section.

Tie-breaking heuristic One option when classification fails due to the false positives, is to just not return back an answer; this may be an acceptable choice given the system inherently trades-off some amount of accuracy in order to scale better. We can also choose to resolve such conflicts using the following *tie-breaking heuristic.* When a packet can potentially match many groups, we report the latency value of the group with the largest number of packets among all conflicting groups. For identifying this, we assume we can store *running* packet counts for each group in an extra counter. This approach now can introduce *false classification* because the decisions can be wrong. But, we observed that this heuristic can work well when the distribution of the cardinalities of BFs is skewed (*e.g.*, long tailed, heavy tailed), and can improve accuracy in many cases. However, as we mention in §4, the result of a query will be explicitly tagged so that the application which uses this data can be informed about the 'guess'ing nature of the answer.

3.4 Analysis of PLS

In this section, we discuss why simple hash table cannot scale in terms of space requirement while achieving $O(1)$ insert and lookup, and analyze the dependence of collision performance of the proposed data structures on storage dimensioning.

3.4.1 Hash table

While hash tables are typically simple, at a minimum they require the packet hash (32 bits) and group id (6 bits for 50 centers), thus requiring at least 38 bits per packet. Collision avoidance schemes present a further challenge for scaling. Thus, in order to perform a comparison with a BF, we consider a simpler hash table with no collision avoidance scheme, in which the packet digest is used to address a memory location in which the group id is stored (collisions will override the group id). For our analysis we consider n packets whose digest values are distributed independently and uniformly across m locations. Following §3.3.4, the false classification probability p_{FC} is proportion of packets allocated to already occupied locations: $p_{\text{FC}} = 1 - \frac{m}{n}(1 - (1 - 1/m)^n)$; see e.g. Section 3.3.2 of [22]. Although the required capacity m is not given as an explicit function of a target p_{FC}, we have the approximation $p_{\text{FC}} \approx n/(2m)$ when $n \ll m$. For example, when $p_{\text{FC}} = 0.02$ (a median false classification rate that SVBF achieves in §5.3) then $m = 24.6n$. Considering $k = 50$, each bucket is 6 bits. Then, $m/n = 147$ bits/packet, even higher than the regular hash tables. Thus, this simple variant does not scale.

3.4.2 Collision analysis & storage dimensions

We now analyze the frequencies of classification failures due to storage collisions for queries on the PBF, COMB and SVBF data structures. First, it is convenient to identify a generic collision analysis that applies to each storage method. Following the terminology of Section 3.3, the *false positive* probability p_{FP} denotes a probability that a given set

Data structure	#Hash functions	Capacity (m/n)	Insertion	Lookup	Note
PBF	$h_{\text{PBF}} = -\log_2(1-(1-p_{\text{CF}})^{1/(k-1)})$	$\geq h_{\text{PBF}}/\log 2$	h_{PBF}	$k \times h_{\text{PBF}}$	lookup can be parallelized
COMB	$h_{\text{COMB}} = -\log_2(1-(1-p_{\text{CF}})^{1/(f-\theta)})$	$\theta \times h_{\text{COMB}}/\log 2$	$\theta \times h_{\text{COMB}}$	$f \times h_{\text{COMB}}$	random access for lookup
SVBF	$h_{\text{SVBF}} = -\log_2(1-(1-p_{\text{CF}})^{1/(k-1)})$	$h_{\text{SVBF}}/\log 2$	h_{SVBF}	$(\lceil k/w \rceil + 1) \times h_{\text{SVBF}}$	serial burst read in the unit of word

Table 2: Complexity of storage data structures for single port memory. w is the size of memory word. $\log$ is natural log.

of storage locations pertaining to a single delay group are occupied. Then, *classification failure* for a packet in delay group i occurs unless it has no false positive in any other delay group j: $p_{\text{CF}}^{(i)} = 1 - \prod_{j \neq i}(1 - p_{\text{FP}}^{(j)})$.

Consider now specifically a BF with m locations and h hash functions. We assume an independent hash digest distribution over all packets. For simplicity, we assume that the query packet is mapped by the hash functions to h distinct locations[1]. As is well known, the false positive probability that a set of h bits are all set after the insertion of n background objects is $p_{\text{FP}} = p(m,n,h) = (1-(1-1/m)^{nh})^h$.

Collisions in PBF. Each delay group i has capacity m_i ($\sum_i m_i = m$) and n_i background packets allocated to it ($\sum_i n_i = n$). For a query on a packet in delay group i, classification failure occurs unless there is no false positive in any other delay group. Thus, averaging over all n packets in their respective delay groups, we have

$$p_{\text{CF}} = 1 - \sum_j \frac{n_j}{n} \prod_{i \neq j}(1 - p(m_i, n_i, h)) \tag{1}$$

Since the operational allocations m_i and n_i are not known in advance, for design purposes one would assume uniformity, in which case (1) reduces to (3) below.

Collisions in COMB. All delay group locations for the query flow may be set by any of the background flows. Since COMB uses θ bits to denote a group id in a code, it is equivalent to virtually put $\theta \cdot n$ items into m locations. Hence

$$p_{\text{CF}} = 1 - (1 - p(m, \theta \cdot n, h))^{f-\theta} \tag{2}$$

Collisions in SVBF. All delay group locations for the query flow may be set by any of the background flows. Hence

$$p_{\text{CF}} = 1 - (1 - p(m, n, h))^{k-1} \tag{3}$$

When all BF sizes m_i are equal, a standard convexity argument shows that $p_{\text{CF}}(\text{SVBF}) \leq p_{\text{CF}}(\text{PBF})$ for any $\{n_i\}$.

Storage Dimensioning. We use the foregoing analysis to show how to dimension SVBF for given target classification failure rate p_{CF}. For large, n, $p(m,n,h) \approx q(n/m, h)$ where $q(\alpha, h) = (1 - e^{-\alpha h})^h$. As is well known, $\alpha \mapsto q(\alpha, h)$ is minimized at when $h = \alpha^{-1}\log(2)$, in which case, $p_{\text{CF}}(\text{SVBF}) = 1 - (1 - 2^{-h})^{k-1}$. Thus, given a target p_{CF} of $\varepsilon > 0$, we must choose h and $\alpha^{-1}\log 2$ to be bounded below by $-\log_2(1-(1-\varepsilon)^{1/(k-1)})$. The lower bound for h and upper bound for α compatible with two possible target classification failure rates $\varepsilon = 10^{-1}$ and 10^{-5} are displayed as a function of the number k of delay groups in Figure 4. Observe that, due to the logarithmic dependence, after an initial phase, the curves are relatively flat as a function of k. They do not depend very strongly on the target rate: decreasing ε by 4 orders of magnitude changes the bounds by only about half an order of magnitude. Table 2 provides overview of insert and lookup complexities and storage requirements of the three data structures. Given $\varepsilon = 10^{-1}$, parameter tuning examples are shown in Table 1.

[1]This happens with probability $m^{-h}m!/(m-h)! \geq 1 - h^2/(2m)$

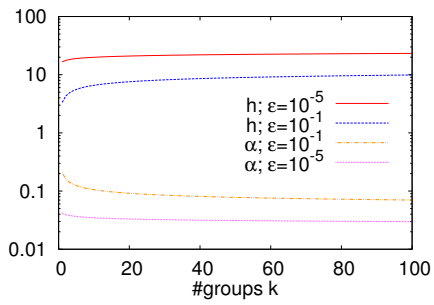


Figure 4: Dimensioning SVBF: lower bound on #hash functions h, and upper bound on load $\alpha = n/m$, as function of #delay groups k, for two target p_{CF} classification failure rates $\varepsilon = 10^{-1}$ and 10^{-5}.

4. LATENCY QUERY INTERFACE

In this section, we describe the packet latency query interface that allows a 'querying client' (henceforth, just client) to query routers for specific packet latency measurements.

Query using packet hash In the very basic query, a client can request a particular router for latency of a given packet identified by the packet hash. This implicitly assumes that the packets are first hashed using the invariant fields in a packet header (*e.g.*, IP addresses, IP id, port numbers) and the packet payload (few bytes is often sufficient [15]) before inserting into the SVBF. We also assume that the client knows that the path taken by the packet, otherwise, the client needs to ask all the routers in the network which may increase the number of bogus queries. We assume that it will be possible to determine this based on the forwarding tables.

The switch then performs a lookup operation of the packet in the SVBF to return back the latency estimate to the client. Because of classification failure possibility in SVBF, we return latency estimates with a 2-bit type that identifies one of three types: (1) `Match` that indicates that the packet was uniquely identified in the SVBF. (2) `Multi-Match` indicating that multiple matches were reported, but the latency estimate corresponds to the answer using the tie-breaking heuristic we discussed in §3.3.4. (3) `No-Match` that indicates the packet's latency estimate could not be located.

If the host wishes to obtain flow-level (or any other aggregate) latency statistics, it needs to send all packet digests of a particular flow of interest to the particular switch/router it believes the packet may have traversed along the path from the source to the destination. Sending one packet for querying each packet to the switch may lead to too many packets. Luckily, packet digests could be easily batched in one query message (about 375 32-bit labels can be embedded within one 1500 byte query packet). Since packet hashes are random d-byte strings, this is the only way to form the aggregate unfortunately. We can reduce the query bandwidth by potentially querying only a sample corresponding to the aggregate as opposed to all the packets, that may represent a trade-off between query bandwidth and accuracy. While exploring this trade-off is outside the scope of this paper, we briefly describe an idea next that has the potential to reduce the query bandwidth significantly.

Query using flow key and IP identifier We can reduce query bandwidth overhead with the help of a range search capability. Since packet hashes do not lend themselves to this range search easily, we consider an alternate scheme. Instead of storing the packet hash, we store the concatenation of the packet's flow key and the IP identifier (IPID) field. The benefit of using IPID field is that it is incremented linearly for each packet transmitted by end systems, allowing us to support flow-level (or sub-flow level) queries quite easily. The implementation of IPID field can be error-prone in some hosts, however. But, we assume that in datacenter environments, because hosts are controlled by one single owner, they can detect any anomalies or fix implementation bugs through patches, if any.

The client in this case can query packets using the tuple $(f_k, [IPid_i, IPid_j])$. Note that this does not stipulate that all packets that belong to a flow will need to have contiguous IDs. But since TCP transmits packets in bursts usually, we can break down a flow in to several tuples

$$f_k = (f_k, [IPid_{i1}, IPid_{i2}]), (f_k, [IPid_{i3}, IPid_{i4}]), \ldots,$$

and chain them together in one message. This reduces the query bandwidth significantly, almost by a factor of 10× in our evaluation (§5.4). The receiving router will sequentially query all packets $(f_k, IPid_{i1})$, $(f_k, IPid_{i1+1})$, ..., $(f_i, IPid_{i2})$. Further, we can make the query interface specify whether it wants the individual latency values or the aggregate values, so that the router can send either individual packet latency values or aggregate them in its response. This will also reduce the response overhead significantly.

Query timing For both types of queries, we need the client to mention a rough time of the packet as part of the query, so that the router can lookup the appropriate SVBF data structure corresponding to the time when the packet may have gone through the router. Given the fact that PLS resets the SVBF's every epoch, it is important to make sure that the previous epoch and the next epoch are also queried for the packet in order to make sure there are no fringe effects, i.e, the timing is close to the start/end of an epoch and it may lie before or after the epoch.

Querying clients Our architecture is largely oblivious to who originates the query. In one scenario, we could envision end hosts could be the clients in some environments. For example, an end host that is running a low-latency trading application or a high-performance computing application, whenever it detects packet delays exceed some level, may originate a query packet to determine which router is responsible for the higher delay. Similarly, we can consider private datacenter owners such as Google, Microsoft, etc., that may want to debug their systems may provide this ability for individual hosts to query routers periodically for obtaining latency statistics.

We can also consider public cloud environments such as Amazon EC2 where customers may demand certain SLAs on network performance. We could imagine the cloud provider installing a debugging stub-module within the host hypervisor (similar to other recent works [32]) that essentially, at the signal of a management host controlled by the network operator, can start storing each packet's hash that matches a given hurting application, or a hurting customer. It can then query these packets along the route to its destination to determine its latency. In this case, it makes sense to put this stub module within the hypervisor since it is the one that knows which packets are going out of its system; the management host cannot possibly know how to query for the packets since it does not know either the packet hash or the IP id sequence (for the compressed query).

In the first usage scenario, we essentially trust the end host to not overwhelm the switch by injecting too many queries. This is possible in a tightly controlled datacenter or cluster environment, but may not be, for example, possible in a public cloud environment such as Amazon EC2 (the second scenario). In such cases, we need some other protection mechanisms (*e.g.*, charging models, rate limiting) to ensure the number of queries to switches does not exceed some limit.

5. EVALUATION

In this section, we evaluate the practicality of our MAPLE architecture. Specifically, our experiments are designed to answer the following questions. (1) How do the different clustering algorithms perform ? (2) How do the various data structures we discussed in §3.3 compare in terms of their accuracy for a given storage budget ? (3) How efficient is the query interface in terms of latency estimates of arbitrary aggregates, bandwidth reduction and inaccuracies in query timing ? How does it compare with previous approaches such as RLI ? We first describe our experimental setup before answering these questions.

5.1 Experimental setup

While we envision the eventual deployment to be in the form of a hardware prototype, for the purposes of evaluation, we prototyped various pieces, notably the streaming clus-

tering algorithm and the storage data structure, of MAPLE in software. We implemented the online portion of the k-medians algorithm from scratch (about 300 lines of C++ code), while we used the C clustering library [4] for the offline part. However, the library had to be modified to support clustering data with weight (*i.e.*, count of entries clustered to a candidate center at the online stage). For most of the experiments, we use 50 centers ($k = 50$) that, as we shall show, represents a good balance between accuracy and complexity.

In our setup, we feed several packet traces (real router as well as using synthetic queuing models) into the software prototype to study its efficacy. We can however easily replace the packet traces with live traffic in our environment. For the most part, we kept our evaluation setup very similar to prior work [24]. We also used the same traces—a tier-1 trace (*BB*) collected at an OC-192 link and a real router trace (*RR*)—as the authors of [24] to facilitate a fair comparison with prior work. While [7] have recently published some data center traces, they belong to a university data center edge router with data rates of 2-3Mbps and is not a sufficient workload for testing the scalability of our architecture.

The *BB* trace we used in our setup has about 22.4M packets in a period of 60s. We divide the 60s period into 60 measurement epochs each with 1s duration. Recall that our architecture operates in epochs and freezes the storage for lookup after every epoch. Thus, average number of packets over 60 epochs is 373,850. The minimum and maximum numbers of packets are 357,912 and 404,452 respectively. We find about 40 thousand hosts and 4.2M flows (considering same flow key across two epochs as two different flows) on average in each epoch. We also conducted experiments using an *RR* trace set that contains two different traces; one trace has 2.6 million packets for 5 minutes achieving 53% utilization of an OC-3 link, and the other trace has 4.0 million packets during the same period (88% link utilization). The traffic source is artificial in that, it is generated by Harpoon traffic generator, but all packets were subject to latency factors in a real router. Qualitatively, we found consistent results across both *BB* and *RR* traces and hence we do not discuss the results on *RR* traces any further.

Following the setup in [24], we subject packets to a simple queueing model with open-loop RED queue management strategy with parameters configured similar to the setup in [24]. We configure the packet processing time in terms of byte/second and queue length in our queueing model. Real packet lengths and inter-arrival distribution govern dynamics of packet delay values and losses.

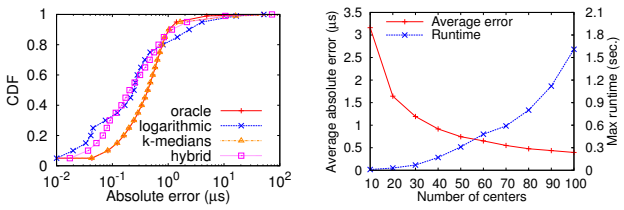
5.2 Performance of clustering algorithms

We first compare static (logarithmic), dynamic (pipelined k-medians) and the hybrid strategy that combines the two. For reference we also include a *hypothetical* non-pipelined k-medians (called *oracle*) approach, that essentially runs the k-medians on the data directly, determines the centers, and then clusters the packets into these centers. In all algorithms, we assume a perfect data structure for storing the approximate delays, i.e., no Bloom filters to introduce any interference. This gives us a baseline for comparison. As mentioned before, we compare these schemes assuming $k = 50$ centers. While we conduct experiments with three traces of different link utilization scenarios to comprehensively understand the tradeoffs, we omit showing all the curves due to space limitations. We show the absolute error CDF for only the moderate utilization case in Figure 5(a).

In our experiments, we observe that oracle achieves the smallest absolute error at higher quartiles among all methods across all link utilization scenarios as it minimizes the summation of absolute distance between entries and their closest centers, which is exactly the absolute error. Since its objective is to decrease the absolute error, it may allocate centers that may increase the relative error for some packets, particularly the low-latency packets. We observe a similar trend in the k-medians clustering method as well. Logarithmic clustering generally achieves higher accuracy than other methods in terms of relative error except low utilization scenario because its centers are placed at equal distances within each sub-range (*e.g.*, 1-10μs and 10-100μs). In low utilization scenario, there is a larger fraction of packets whose delays are far smaller than the nearest center leading to worse accuracy. Comparatively, the other schemes adjust to this trend quickly and place more centers close to where the actual delays are, leading to better accuracy.

k-medians clustering method has the similar performance that the oracle has in terms of both absolute and relative errors under low, moderate utilizations, and even the lower quartiles of the higher utilizations. At higher quartiles of higher utilization scenario, the k-medians clustering method is worse than that of oracle because of the inherent variations across epochs; this is in essence the price we pay for an online clustering algorithm. Finally, we observe that the hybrid clustering approach balances both absolute and relative errors by inheriting the good properties of static and dynamic center determination approaches. For instance, in Figure 5(a), we can see how the hybrid scheme inherits the better accuracy of logarithmic approach at lower quartiles and better accuracy of k-medians at the higher quartiles. Henceforth, unless otherwise mentioned, we use the hybrid scheme throughout the rest of the paper.

Impact of packet sampling. In §3.2, we discussed that we employ packet sampling in the clustering phase to reduce the processing overhead. We now study the impact of varying the sampling rate on the accuracy of the per-packet latency estimates. In our experiments, we found virtually no difference between the different absolute error CDFs (and hence, refrain from showing the actual plots) between the sampled and unsampled variants. One could imagine this happens because most packet delays are clustered to statically chosen centers instead of those close to the k-medians. However, this is not the case, as only 53% and 1.5% packets are clustered to those static centers at high and low link utilizations



(a) Clustering schemes

(b) Trade-offs in determining centers

Figure 5: Comparing different clustering schemes and exploring trade-off between average error and maximum running time for offline stage.

respectively. The actual reason is that the k centers output by the algorithm are similar even when we sample packets. We observe that the cosine similarity (defined as $\cos\theta = \frac{A \cdot B}{|A||B|}$) between two vectors A and B of k centers output by sampled and unsampled k-medians algorithm is over 0.98 for sampling rates as low as 1%.

Number of centers vs. running time. The running time of the offline algorithm depends on the final number of centers required, i.e., k, and the number of candidate centers output by the online clustering stage. As k increases, the running time of the algorithm increases, but the resulting error also decreases. This is the main trade-off involved in choosing an appropriate value of k. Figure 5(b) explores this trade-off as k is increased from 10 to 100. (We only plot the curve corresponding to the sampling rate 0.1 here, but other sampling rates also exhibited similar trends.) From the plot, we can clearly observe the sweet spot that represents a reasonable trade-off between running time and average absolute error is $k = 50$. We can possibly choose up to 70 or 80 centers as well as the running time is less than the epoch interval (1s). Of course, depending on the target platform and computational resources we can expect on the processor, the number of centers may vary. But the existence of this trade-off implies we can easily determine the value of k appropriate for the target platform.

5.3 Comparison of data structures for PLS

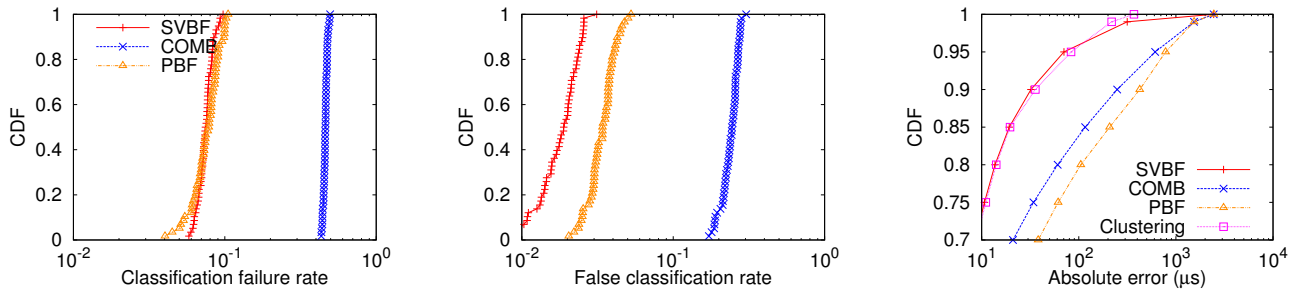
Next, we compare the performance of various data structures for PLS—SVBF, PBF and COMB. For fair comparisons, we configure all of these with the same amount of memory, 5Mbits in total. It is not easy to fix the amount of memory in PBF since the total memory needs to be explicitly partitioned across all the BFs. We split the total memory across each BF proportional to the number of packets that are mapped to a given BF (according to the frequency counts two epochs back). For the others, we can derive the optimal parameter configurations, such as number of hash functions, for different data structures using the formulae in §3.4.2. For this memory, the theoretical analysis suggests using 9 hash functions for SVBF and PBF. For COMB, there are two other parameters, θ (number of bits that need to be set in the group code) and f (length of the code). Out of feasible combinations to support $k = 50$ groups, we choose the configuration with f=8 and θ=3 that has the smallest lookup time (that is proportional to f). By fixing these parameters, the number of hash functions per bit h needs to be set to 3 according to the analysis in §3.4.2. Thus, all in all, we ensured that the comparisons are as fair as possible between the various schemes.

In our comparisons, we mainly study the classification failure rate, false classification rate, and finally the impact of these on the accuracy of latency estimates. We search latencies of all packets for 58 epochs (the first 2 epochs are used for clustering only). For false classification rate, we use the tie-breaking heuristic described in §3.3.4 and compute the rate at which the heuristic leads to an incorrect answer.

Classification failure and false classification rates. We show the classification failure rate of each data structure in Figure 6(a). As we can observe, SVBF and PBF (as expected) achieve least classification failure rate of about 10% at most across all epochs, while COMB obtains 50% at most—almost 5× higher than SVBF. Note that for PBF due to the fact that for a given BF, the number of packets may exceed the capacity, we observed almost 24% of packets could not be admitted altogether (false negatives). Applying the tie-breaking heuristic results in a false classification rate that is lower than the classification failure rate, but not by much. Still, as we can see from the Figure 6(b), the median false classification rate of COMB is almost 12× higher than SVBF. This shows the efficacy of our SVBF compared to existing data structures such as COMB. As shown in Table 1, COMB requires almost twice the number of bits per packet to achieve the classification failure rates as SVBF. We next study how this decrease in classification failure effects the actual delay distribution.

Accuracy of per-packet latency estimation. Figure 6(c) shows comparison results in terms of absolute errors of per-packet latency estimates. In addition to these data structures, there is an additional curve titled 'Clustering' that essentially assumes a perfect data structure, but does not use the refined latency estimates (described in §3.3) using the observed mean of the data packets. (We can always plot that too, and that would strictly be better than the rest, but we chose this as a nice reference point to see the effects of the refinement.)

We show mainly the upper quartile in this graph where the difference is the most pronounced. Clearly, at lower quartiles, either SVBF or COMB would return the same (correct) group id if the packet is not misclassified; it is only for the mis-classified packets that the accuracy is likely to be worse since the tie-breaking heuristic may pick the wrong latency estimate for the packet. (If we choose not to report them, then they will be counted as false negatives.) We can notice that COMB and PBF suffer from much higher discrepancies as early as the 70%ile onwards, while in contrast we can see that the Clustering and SVBF have an absolute error that is significantly lower in comparison. For example the 85%ile



(a) Classification failure (b) False classification (c) Estimation accuracy

Figure 6: Analysis of classification failure, false classification, and estimation accuracy.

absolute error for COMB is close to $116\mu s$ while SVBF has an error of 19μ at the same percentile. From the figure, we can see that not until almost the 98%ile onward do we see any difference between Clustering and SVBF.

5.4 Query interface

Since our architecture can support querying any packet, it can allow the querying host to compute aggregate statistics across arbitrary traffic sub-populations.

Accuracy of aggregate statistics In the first experiment, we verify the accuracy of obtained aggregate statistics (by querying packets that belong to that aggregate) at different levels–sub-flow, flow, host, and prefix/16. By performing flow-level aggregation, i.e., by grouping packets with the same flow key, our architecture achieves similar functionality as previous work RLI [24]. However, perhaps more unique to our architecture, due to the fact that it stores measurements on a per-packet basis, we can choose to aggregate at sub-flow-level, while RLI cannot easily achieve this. We compute the average delay of 10 consecutive packets within the same flow key (among large flows whose size is more than 10 packets) as the sub-flow average delay. Such a feature could be useful, for instance, to understand which set of packets within a large flow are exhibiting abnormal latencies.

Figure 7(a) shows the aggregate statistics in terms of relative error for the high link utilization scenario. We also draw a curve for packet latencies as a reference curve. From the figure, we observe that as aggregation level becomes higher, relative error reduces. Latency estimates at sub-flow level, however, has the least relative errors. This is not inconsistent, since many flow/host/prefix-level statistics, although aggregated with packets within a given epoch, are computed with only a single packet (46% flows, 41% hosts, and 13% prefix/16), while sub-flow statistics are computed for flows that at least have 10 packets. Thus, sub-flow latency estimates get more influence on canceling individual errors out by aggregation. Specifically, median relative error is 5.5% at packet level, 3.9% at flow level, 3.6% at host level, 2.1% at prefix/16 level and 1.9% at sub-flow level. Similar trends are found under low and moderate utilization scenarios.

In terms of absolute errors, we observe that prefix/16 average latency is more accurate than other aggregation levels, at low link utilization. As link utilization increases, however, we find little difference in absolute error among all four aggregation levels. We omit graphs for brevity, but we observe a 95-percentile absolute error of less than $0.05\mu s$, $2\mu s$ and $55\mu s$ across all four aggregation levels.

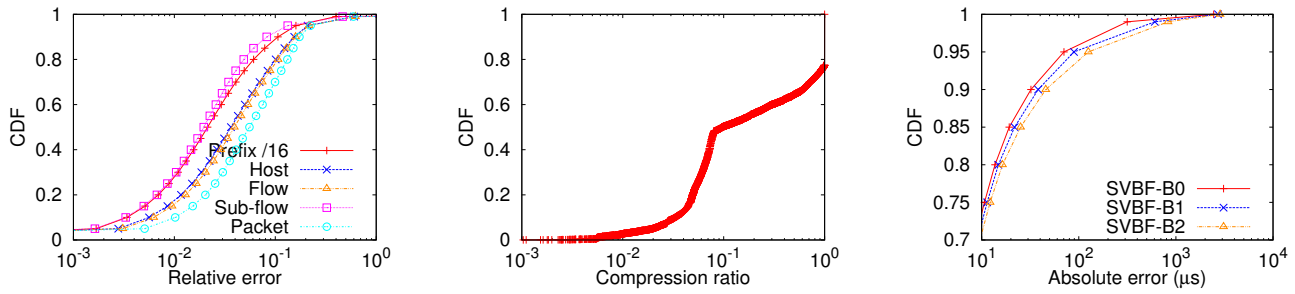
Compression of query bandwidth with IP ids. We study the query bandwidth saving using the IPID idea outlined in §4. For each flow within an epoch, we compare the bandwidth of range query messages with individual packet queries. Figure 7(b) shows CDF of compression ratio of flows having more than 100 packets for 60 epochs with *BB* trace. From the figure, we observe that we can achieve a median compression ratio of about 10%—10× less bandwidth than the naive—while the total compression ratio if we want query for all flows is about 25%.

Impact of inaccurate query timing We evaluate the impact of inaccurate query timing when clients issue per-packet latencies. In the experiments, all packet queries in an epoch i are asked to a SVBF of that epoch (true SVBF) and additional b number of SVBFs of previous epochs $i - b$ (bogus SVBFs). Multiple matches are resolved using the same tie-breaking heuristic. In Figure 7(c), we show the results for 2 bogus SVBFs. Clearly, as the number of epochs considered increases, the accuracy decreases slightly. For instance, 95%ile absolute error shifts from 70 to 89 to $125\mu s$ as bogus SVBFs are added, but the 75%ile errors are not that impacted, increasing the error from 10.7 to $12.2\mu s$.

5.5 Comparison with prior architecture

We now compare our MAPLE architecture with RLI [24], that also averages approximate latencies of packets (obtained via latency interpolation) that belong to a flow. In MAPLE, we use accurate packet latencies (using timestamps) but store them approximately in the SVBF data structure. MAPLE also can use RLI-approximated latency for each packet, but this leads to two sets of approximations (we denote this as MAPLE-RLI). We study these various effects in Figure 8. (For brevity, we only show the high utilization curve; the trends for the other two utilizations were similar.)

From Figure 8, we make two observations from the fig-



(a) Accuracy of aggregates (b) Query bandwidth compression (c) Impact of query timing

Figure 7: Average latency statistics at different aggregation levels, query message compression ratio about flows having more than 100 packets with IPID field, and impact of query timing using high utilization scenario.

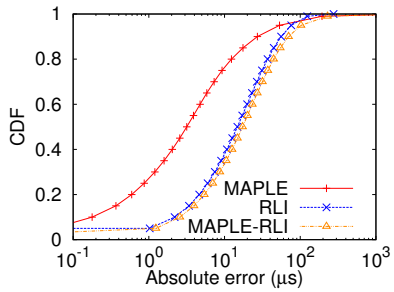


Figure 8: Comparison of flow estimates with RLI.

ure. First, there is little difference in absolute error between MAPLE-RLI and RLI, while RLI has slightly higher accuracy than MAPLE-RLI, which is expected. Specifically, under high utilization scenario, RLI has 76μs absolute error at 95 percentile, but MAPLE-RLI has 102μs at the same percentile. Median error by RLI is 15μs and the error of MAPLE-RLI is 17μs. The second observation is that MAPLE (with true latencies) achieves much higher (half to one order of magnitude higher) accuracy than RLI. For instance, about 90% flows have less than 1μs absolute error with MAPLE, but RLI only has 50% flows with such absolute error under moderate utilization scenario (not shown for brevity). Put differently, in Figure 8, MAPLE achieves 5$\times$ less median error than RLI. This shows that MAPLE, if implemented, may provide more accurate latency estimations than RLI, even with the approximations in the storage data structure.

6. IMPLEMENTATION

We envision that the streaming k medians algorithm will be implemented in software. We assume there is an additional processor (or core) devoted to implementing this architecture. We assume that this core will perform the streaming k-medians on the sampled data. There have been some prior efforts [26] on implementing k-medians directly in hardware that we can also leverage. In environments where there is not enough processing capacity, we can rely on the static clustering approach we discussed in §3. This will however yield worse accuracy than the hybrid estimator.

The actual storage data structure, SVBF, outlined in Figure 2 will need to be implemented in hardware in high-speed SRAM. Bloom filters in general require simple hashing operations and updating bit maps and thus are amenable to high-speed implementations (see [34] for example). For implementing the hash functions, we can use the H3 [31] hash functions or the BOB [21] hash functions that are amenable to easy hardware implementations. We need to also maintain two extra counters per-center, one for tracking number, and the other for sum of delays of all packets that map to a given center. These counters will enable the refined latency estimate heuristic (in §3.3) and the tie-breaking heuristic (in §3.3.4). The SVBF data structure needs to be flushed every epoch to an off-chip storage, which can be either DRAM or SSDs. Along with each SVBF, the associated k centers for that particular epoch need to be stored. For smaller k, this is only a small amount of extra storage. Depending on the technological constraints such as the amount of available high-speed memory and link speeds, the epoch size could be determined.

Assuming an OC-192 interface, we have roughly 5 million packets per second, for which we will require about 60 Mbits of memory per second (assuming 12 bits/pkt). Of course, this is assuming the interface is running at full capacity, which is often not the case. Thus, if we assume 10% utilization, we only require 6 Mbits of memory per second. Thus, 16 GB of DRAM (which is commodity today) could be used to store packet latencies for almost 40 minutes. Flash memory densities are even higher; today 256 GB SSDs are possible which will enable storing packet latency state for about 10 hours, which is enough time for network operators to debug and process the information.

Queries will need to be handled in software. For each query, depending on the approximate time of the packet in the query, the appropriate SVBF (and the two neighbors just in case) will be queried by the processor (or core) (perhaps shared with the k-medians implementation). Only the words corresponding to the hash indexes will need to be fetched from the secondary memory (SSD or DRAM), which are then looked up according to the algorithm outlined in §3.3.3.

7. RELATED WORK

There exists a lot of research in measuring per-hop latencies, although in the wide-area context, where ISPs typically rely on injecting active probes and obtaining link or hop latency statistics using tomographic approaches [12, 14, 35]. These approaches do not satisfy any of our high-fidelity measurement requirements in §2.1 and thus we need high-fidelity passive measurement mechanisms. In this regard, we already discussed three prior approaches relevant to our—LDA [23], RLI [24] and Consistent NetFlow [25].

The idea of storing packet-level information has been pursued in other prior contexts; trajectory sampling for identifying packet trajectories in [15] and SPIE for IP traceback in [33]. Neither provides latency estimates unfortunately, although trajectory sampling could be augmented with a timestamp, but only a small number of packets are sampled at each router (see [23, 24] for comparison of these approaches with trajectory sampling). SPIE on the other hand stores only packets and not their associated timestamps; thus, a simple Bloom filter was sufficient there, while we needed the clustering and SVBF in our setting.

The idea of 'in-band' diagnosis was proposed in NetReplay [6] and Orchid [30]. NetReplay proposes the idea of replaying packets to collect feedback from the network. Orchid [30] also propose the idea of in-band network troubleshooting, where packets collect feedback from routers along the path. Our approach, however is more focused on estimating, storing and retrieving packet-level latency measurements, and is complementary to these approaches.

Song *et al.* propose fast hash table [34] to provide constant lookup time by exploiting counting bloom filter. Fast hash table does not address large space requirement because of an extra counting bloom filter and the need to store both packet digest and its delay. Supporting membership check across multiple groups is non-trivial for Bloom filter. Several data structures [9, 11, 17] have been proposed to address this problem. COMB [17] is a multi-group membership check data structure that is highly relevant to our work and hence, we discussed this before in §3.3.

8. CONCLUSION

This paper proposed a scalable and flexible measurement architecture called MAPLE. The core of the architecture consists of two novel mechanisms; a streaming clustering algorithm to cluster packet latencies into small number of latency clusters in a streaming fashion, and a data structure called SVBF to store packet latencies efficiently in a router. In addition, it provides a flexible query interface for network operators to query the latency of individual packets. Together, the architecture provides both fine-grained as well as flexible latency measurements to help network operators manage low-latency applications efficiently. Our evaluations using a software prototype indicate that our architecture can scale efficiently both in terms of storage needs as well as in terms of query bandwidth.

9. REFERENCES

[1] Cut-through and store-and-forward ethernet switching for low-latency environments. `http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-465436.html`.

[2] FocalPoint TDM Support. `http://www.fulcrummicro.com/product_library/applications/TDM_App_Note.pdf`.

[3] Sprint unveils SLAs for Internet access, Latency, Packet loss. `http://www.crn.com/news/channel-programs/18827630/sprint-unveils-slas-for-internet-access-latency-packet-loss.htm;jsessionid=H9Xzdo9g+LhlF-2Sa89Big**.ecappj02`.

[4] The C Clustering Library. `http://bonsai.hgc.jp/~mdehoon/software/cluster/software.htm`.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM*, 2010.

[6] A. Anand and A. Akella. NetReplay: a new network primitive. *ACM SIGMETRICS Performance Evaluation Review*, 37, 2010.

[7] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM/USENIX IMC*, 2010.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.

[9] F. Chang, F. Chang, and W. chang Feng. Approximate Caches for Packet Classification. In *IEEE INFOCOM*, 2004.

[10] M. Charikar, L. O'Callaghan, and R. Panigrahy. Better Streaming Algorithms for Clustering Problems. In *ACM STOC*, 2003.

[11] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *ACM SODA*, 2004.

[12] Y. Chen, D. Bindel, H. Song, and R. H. Katz. An Algebraic Approach to Practical and Scalable Overlay Network Monitoring. In *ACM SIGCOMM*, 2004.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[14] N. Duffield. Simple network performance tomography. In *ACM/USENIX IMC*, 2003.

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

[16] J. Eidson and K. Lee. IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *Sensors for Industry Conference, 2002. 2nd ISA/IEEE*, 2002.

[17] F. Hao, M. Kodialam, T. Lakshman, and H. Song. Fast multiset membership testing using combinatorial bloom filters. In *IEEE Infocom*, 2009.

[18] P. Indyk. A Sublinear Time Approximation Scheme for Clustering in Metric Spaces. In *IEEE FOCS*, 1999.

[19] P. Indyk. Sublinear Time Algorithms for Metric Space Problems. In *ACM STOC*, 1999.

[20] A. K. Jain and R. C. Dubes. Algorithms for Clustering Data: Prentice-Hall, 1981.

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

[22] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981.

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

[24] M. Lee, N. Duffield, and R. R. Kompella. Not All Microseconds are Equal: Fine-Grained Per-Flow Measurements with Reference Latency Interpolation. In *ACM SIGCOMM*, 2010.

[25] M. Lee, N. Duffield, and R. R. Kompella. Two Samples are Enough: Opportunistic Flow-level latency estimation using Netflow. In *IEEE Infocom*, 2010.

[26] M. Leeser, J. Theiler, M. Estlick, and J. Szymanski. Design tradeoffs in a hardware implementation of the k-means clustering algorithm. In *Sensor Array and Multichannel Signal Processing Workshop*, 2000.

[27] S. Lloyd. Least squares quantization in PCM. *Information Theory, IEEE Transactions on*, 28(2):129–137, Mar. 1982.

[28] R. Martin. Wall street's quest to process data at the speed of light. `http://www.informationweek.com/news/infrastructure/showArticle.jhtml?articleID=199200297`.

[29] A. Meyerson. Online Facility Location. In *IEEE FOCS*, 2001.

[30] M. Motiwala, A. Bavier, and N. Feamster. Network troubleshooting: An in-band approach. In *NSDI*, 2007.

[31] M. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient hardware hashing functions for high performance computers. *IEEE Transactions on Computers*, 46(12), Dec. 1997.

[32] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: performance isolation for cloud datacenter networks. In *USENIX HotCloud*, 2010.

[33] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, B. Schwartz, S. T. Kent, and W. T. Strayer. Single-packet IP traceback. *IEEE/ACM Transactions on Networking (ToN)*, 10, 2002.

[34] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In *ACM SIGCOMM*, 2005.

[35] Y. Zhao, Y. Chen, and D. Bindel. Towards unbiased end-to-end network diagnosis. In *ACM SIGCOMM*, 2006.