Computing Bottleneck Structures at Scale for High-Precision Network Performance Analysis

Noah Amsel, Jordi Ros-Giralt, Sruthi Yellamraju, James Ezick, Brendan von Hofe, Alison Ryan, Richard Lethin
(amsel, giralt, yellamraju, ezick, vonhofe, ryan, lethin)@reservoir.com
Reservoir Labs, 632 Broadway, Suite 803 New York, New York 10012, USA

***Abstract*—The Theory of Bottleneck Structures is a recently-developed framework for studying the performance of data networks. It describes how local perturbations in one part of the network propagate and interact with others. This framework is a powerful analytical tool that allows network operators to make accurate predictions about network behavior and thereby optimize performance. Previous work implemented a software package for bottleneck structure analysis, but applied it only to toy examples. In this work, we introduce the first software package capable of scaling bottleneck structure analysis to production-size networks. We benchmark our system using logs from ESnet, the Department of Energy's high-performance data network that connects research institutions in the U.S. Using the previously published tool as a baseline, we demonstrate that our system achieves vastly improved performance, constructing the bottleneck structure graphs in 0.21 s and calculating link derivatives in 0.09 s on average. We also study the asymptotic complexity of our core algorithms, demonstrating good scaling properties and strong agreement with theoretical bounds. These results indicate that our new software package can maintain its fast performance when applied to even larger networks. They also show that our software is efficient enough to analyze rapidly changing networks in real time. Overall, we demonstrate the feasibility of applying bottleneck structure analysis to solve practical problems in large, real-world data networks.**

***Index Terms*—Network, performance, traffic engineering, capacity planning, bottleneck structure, benchmark, congestion control**

I. Introduction

Congestion control is an essential component of high-performance data networks that has been intensely researched for decades. The goal of congestion control is to distribute the limited bandwidth of each link in the network among the various data flows that need to traverse it. Congestion control algorithms have a dual mandate of maximizing network utilization while also ensuring fairness among competing flows. The conventional view of this problem assumes that the performance of a flow is solely determined by its bottleneck link—that is, the link in its path that allocates the least bandwidth to it. Standard congestion control algorithms in the TCP protocol such as Reno [1], Cubic [2], and BBR [3] operate at the level of individual flows, the transmission rates of which are set separately by each sender. This perspective makes it difficult to consider the network as a whole, since it hides the complex ripple effects that changes in one part of the network can exert on the other parts.

This work is partially funded by the Department of Energy under contract DE-SC0019523.

The Theory of Bottleneck Structures, introduced in [4], provides a deeper understanding of congestion controlled networks. It describes how the performance of each link and data flow depends on that of the others, forming a latent dependency structure that can be modeled as a directed graph. Armed with this model, network operators can make accurate, quantitative predictions about network behavior, including how local changes like link upgrades, traffic shaping or flow routing will propagate, interact with one another, and affect the performance of the network as a whole. The Theory of Bottleneck Structures can be used to reason about a large variety of network optimization problems, including traffic engineering, congestion control, routing, capacity planning, network design, and resiliency analysis [5].

The goal of this paper is to demonstrate that the insights of the Theory of Bottleneck Structures can be applied at scale to production networks. Previous work introduced a software system that implemented the two core operations of constructing the bottleneck structure graph and computing derivatives of network performance with respect to parameters like link capacities and traffic shapers [5]. However, this system was tested on relatively small networks, and its performance was not benchmarked. In this work, we demonstrate a new high-performance software package designed to scale these two core operations to production-size networks. Using real production NetFlow logs from ESnet—the Department of Energy's high-performance network connecting the US National Laboratory system—we performed extensive benchmarks to compare the two packages and characterize their scalability. We confirm that, with the right implementation, bottleneck structures can be used to analyze large networks in practice, thus unlocking a powerful new framework to understand performance in production environments.

This paper is organized as follows. In Section II, we provide a brief introduction to bottleneck structures and summarize the core algorithms that are the subject of the presented benchmarks. Section III describes the data set and reports the benchmarks for the computation of bottleneck structures (Section III-B) and link gradients (Section III-C). In Section IV we summarize the prior work and Section V provides some notes on integration of the benchmarked algorithms in real production networks. Section VI concludes.

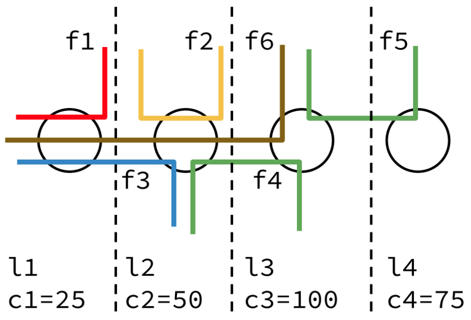


Fig. 1: Network configuration used in Example 1.

II. Theoretical Background and Algorithms

A. *Introduction to Bottleneck Structures*

While describing the mathematics of bottleneck structures is not the focus of this paper, this section provides an example that will give the reader some intuition for the meaning and analytical capabilities of a bottleneck structure.

Example 1. Consider a network consisting of four links $\{l_1, l_2, l_3, l_4\}$ in which there are six active data flows $\{f_1, \ldots, f_6\}$. The capacity of each link $(c_1, \ldots, c_4)$ and the route of each flow is shown in Fig. 1. (We do not consider the network's topology, just the set of links in each flow's route.) The resulting bottleneck structure of this example network is shown in Fig. 2. It is represented by a directed graph in which:

1) There exists one vertex for each flow (plotted in gray) and each link (plotted in white) of the network.
2) a) If flow f is bottlenecked at link l, then there exists a directed edge from l to f.
   b) If flow f traverses link l but is not bottlenecked by it, then there exists a directed edge from f to l.

Intuitively, the bottleneck structure captures the influences that links and flows in the network exert on each other. Consider link 1. Three flows traverse it, and it has a capacity of 25. Thus, it allocates $25/3 = 8\frac{1}{3}$ each to flows 1, 3, and 6. If the capacity of link 1 were to change, the rates of these three flows would change too. This relationship is reflected in the directed edges from node L1 to nodes F1, F3, and F6. Flow 3 also traverses link 2, but since link 2 has more bandwidth available than link 1, flow 3 is not bottlenecked there. The leftover bandwidth not used by flow 3 is picked up by other flows that use link 2—that is, by flow 2 and flow 4. So if flow 3's rate were to change, their rates would be affected too. This relationship is reflected in the directed paths F3 $\rightarrow$ L2 $\rightarrow$ F2 and F3 $\rightarrow$ L2 $\rightarrow$ F4. The reverse is not true. If L2's rate were perturbed by a small amount, F3's performance would not be affected, and indeed, no path from L2 to F3 exists. It has been proven that the performance of a flow f is influenced by the performance of another flow f' if and only if there exists a directed path in the bottleneck structure graph from flow f''s bottleneck link to flow f [4].

The bottleneck structure allows us to easily visualize relationships between network elements. We can also quantify

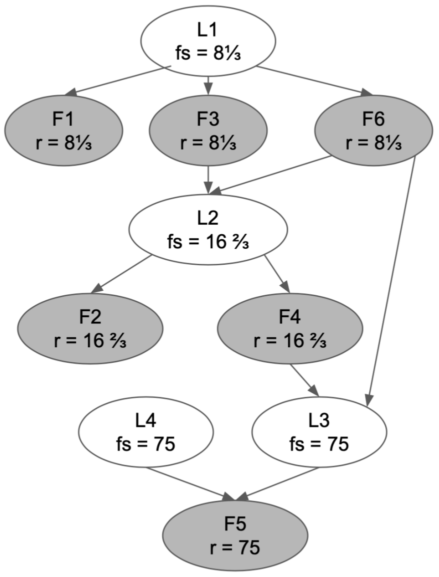


Fig. 2: Bottleneck structure of the network in Example 1.

these relationships. Consider the congestion control algorithm to be a function that takes the network conditions as input and assigns a transmission rate to each flow as output. A key insight stemming from the Theory of Bottleneck Structures [4] is that many seemingly separate questions in network management can be unified under a single quantitative framework by studying the derivatives of this function. [1] For example, letting c_1 be the capacity of link 1 and r_3 be the rate of flow 1, we have

$$\frac{dr_3}{dc_1} = \frac{1}{3}$$

since each additional unit of capacity added at link 1 will be distributed evenly among the three flows which are bottlenecked there. Derivatives with respect to flow rates can also be calculated; they represent, for example, the effect of traffic shaping a flow (that is, artificially reducing its rate) on the performance of another flow. In our experiments however, we always use the capacity c_l of some link l as the independent variable. Derivatives can also be taken of any differentiable function of the rates, not just an individual rate like r_3. In this paper, we take the dependent variable to be the total throughput of the network, that is, the total rate of all its flows:

$$T = \sum_{f \in \mathcal{F}} r_f$$

The derivative $\frac{dT}{dc_l}$ quantifies how much the total throughput of the network would change if link l were given an

[1]The bandwidth allocation function is continuous everywhere, but not technically differentiable. In particular, it is piecewise linear. Thus, while the derivative does not exist at all points, we can study the directional derivative instead. Since the focus of this paper is on benchmarking, without loss of generality we gloss over this technicality and simply use 'derivative' to denote the derivative in the positive direction ($\delta > 0$ rather than $\delta < 0$ in line 2 of Algorithm 3). See [6].

infinitesimally higher capacity.

The Theory of Bottleneck Structures is a somewhat idealized model of network behavior. In our example, we assumed that flow 3 would experience a rate of $8\frac{1}{3}$, but in fact its rate will fluctuate as the congestion control algorithm tries to calibrate it to network conditions, and due to other factors like latency. Nevertheless, it has been shown [4], [5], [7] that the theoretical flow rates predicted by the bottleneck structure model accurately match the actual transmission rates observed in networks that use popular congestion control algorithms like BBR [3] and Cubic [2]. In forthcoming work, we further strengthen these findings by using the G2-Mininet emulation environment [8] to make empirical measurements of flow rates. The Theory of Bottleneck Structures can also be extended; for example, a latent bottleneck structure still exists if a proportional fairness criterion is used to allocate rates instead of max-min fairness. The theory can also be applied to networks that use multipath routing by considering each route to be a separate flow, and optimizing the sum of their bandwidths instead of any individual bandwidth.

B. Applications of Bottleneck Structure Analysis

The scientific community has long relied on high-performance networks to store and analyze massive volumes of data [9]. As the collection of scientific data continues to balloon [10], the importance of designing these networks intelligently and operating them at maximum efficiency will only increase. The analytical power of the Theory of Bottleneck Structures stems from its ability to capture the influences that bottlenecks and flows exert on each other and, in particular, to precisely *quantify* these influences [7]. This ability can be applied to a wide range of networking problems. For example, taking derivatives of the form $\frac{dT}{dc_l}$ is a natural way to study the problem of optimally upgrading the network. The derivative of the total throughput with respect to the capacity of each link reveals which links should be upgraded to have the maximal impact on the overall performance of a network. Other questions in network design and capacity planning can be addressed using similar techniques. The Theory of Bottleneck Structures also sheds light on flow control problems like routing and traffic engineering. For example, if we want to increase the performance of a certain high priority flow and we know which flows are low priority, we can compute derivatives of the high priority flow's rate to determine which of the low priority flows to traffic shape. We can also make precise quantitative predictions of how much this intervention would increase performance. Applications also arise in other areas. For example, determining where a given flow is bottlenecked, who controls that bottleneck link, and how other traffic in the network affects the flow can help in monitoring and managing Service-Level Agreements (SLAs). Future work will describe such applications in greater detail, but few are feasible without high-performance algorithms and software for bottleneck structure analysis. One challenge of analyzing networks in practice is that network conditions change from second to second. The need to analyze networks in real time imposes even stricter performance requirements that previous work has failed to meet.

C. Constructing Bottleneck Structures

This section describes two algorithms for constructing bottleneck structures. The first corresponds to a slightly modified version of the algorithm proposed in [4]. The pseudocode is presented in Algorithm 1 under the name $ComputeBS$.

During each iteration of the main loop, a set of links are resolved, meaning the rates of all flows which traverse them are permanently fixed. This set of links is those whose "fair share value" s_l at that iteration (line 12) is the smallest among all links with which they share a flow (line 13). The rates of all flows traversing link l which have not previously been fixed are set in line 15, and the link and its flows are marked as resolved (line 18 and 19). In addition, the proper directed edges are added to the bottleneck structure graph—from a links to flows which they bottleneck (line 16) and from flows to links that they traverse but that do not bottleneck them (line 17). The algorithm returns the bottleneck structure $\mathcal{G} = \langle V, E \rangle$, the link parameters $\{s_l, \forall l \in \mathcal{L}\}$ and the predicted flow transmission rates $\{r_f, \forall f \in \mathcal{F}\}$.

This procedure is the same as the algorithm proposed in [4], but with additional logic to build the graph representation of the bottleneck structure. Its computational complexity is $O(H \cdot |\mathcal{L}|^2 + |\mathcal{L}| \cdot |\mathcal{F}|)$, where $\mathcal{L}$ is the set of links, $\mathcal{F}$ is the set of flows and H is the maximum number of links traversed by any flow. We leave it as an exercise for the reader to verify that applying $ComputeBS()$ to the network configuration in Fig. 1 results in the bottleneck structure shown in Fig. 2.

Algorithm 1 ComputeBS($\mathcal{N} = \langle \mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\} \rangle$)

```
 1: $\mathcal{L}$ := Set of links in the input network;
 2: $\mathcal{F}$ := Set of flows in the input network;
 3: $\mathcal{F}_l$ := Set of flows going through link $l$;
 4: $c_l$ := Capacity of link $l$;
 5: $\mathcal{B}$ := Set of bottleneck links;
 6: $r_f$ := Rate of flow $f$;
 7: $\mathcal{L}^k$ := Set of unresolved links at iteration $k$;
 8: $\mathcal{C}^k$ := Set of resolved flows at iteration $k$;
 9: $\mathcal{L}^0 = \mathcal{L}; \mathcal{C}^0 = \emptyset; k = 0;$
10: $E = \emptyset;$
11: while $\mathcal{C}^k \neq \mathcal{F}$ do
12:     $s_l^k = (c_l - \sum_{\forall f \in \mathcal{C}^k \cap \mathcal{F}_l} r_f)/|\mathcal{F}_l \setminus \mathcal{C}^k|, \forall l \in \mathcal{L}^k;$
13:     $u_l^k = min\{s_{l'}^k \mid \mathcal{F}_{l'} \cap \mathcal{F}_l \neq \emptyset, \forall l' \in \mathcal{L}^k\}, \forall l \in \mathcal{L}^k;$
14:     for $l \in \mathcal{L}^k, s_l^k = u_l^k$ do
15:         $r_f = s_l^k, \forall f \in \mathcal{F}_l \setminus \mathcal{C}^k;$
16:         $E = E \cup \{(l, f), \forall f \in \mathcal{F}_l \setminus \mathcal{C}^k\};$
17:         $E = E \cup \{(f, l'), \forall f \in \mathcal{F}_l \setminus \mathcal{C}^k$ and $\forall l' \in \mathcal{L}_f \mid s_{l'}^k \neq u_{l'}^k\};$
18:         $\mathcal{L}^k = \mathcal{L}^k \setminus \{l\};$
19:         $\mathcal{C}^k = \mathcal{C}^k \cup \{f, \forall f \in \mathcal{F}_l\};$
20:     end for
21:     $\mathcal{L}^{k+1} = \mathcal{L}^k; \mathcal{C}^{k+1} = \mathcal{C}^k;$
22:     $k = k + 1;$
23: end while
24: $\mathcal{B} = \mathcal{L} \setminus \mathcal{L}^k; \mathcal{V} = \mathcal{B} \cup \mathcal{F}; s_l = s_l^k, \forall l \in \mathcal{B};$
25: $\mathcal{G} = \langle V, E \rangle;$
26: return $\langle \mathcal{G}, \{s_l, \forall l \in \mathcal{L}\}, \{r_f, \forall f \in \mathcal{F}\} \rangle;$
```

We next describe $FastComputeBS$ (Algorithm 2), an improved algorithm for computing bottleneck structures with an asymptotically faster run time than $ComputeBS$. This algorithm resolves links one-by-one, but unlike $ComputeBS$, it stores the links in a heap data structure sorted by the amount of bandwidth they can allocate to flows which traverse them. This allows the algorithm to resolve links in the proper order without searching through the entire set of links at each iteration, effectively skipping the expensive $min\{\}$ computation of Algorithm 1 (line 13). $FastComputeBS$ reduces the asymptotic run time of computing the bottleneck structure to $O(|E| \cdot \log |\mathcal{L}|)$, where $|E|$ is the number of edges in the bottleneck structure and $|\mathcal{L}|$ is the number of links. By definition, there is one edge for each pair of a flow and a link it traverses. Thus, the run time is quasilinear in the size of the input.

Algorithm 2 FastComputeBS($\mathcal{N} = \langle \mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\}\rangle$)

```
 1: $\mathcal{V} = \emptyset$; $E = \emptyset$; $r_f = \infty, \forall f \in \mathcal{F}$;
 2: for $l \in \mathcal{L}$ do
 3:     $a_l = c_l$;      # available capacity
 4:     $s_l = a_l/|\mathcal{F}_l|$;    # fair share
 5:     MinHeapAdd(key = $s_l$, value= $l$);
 6: end for
 7: while $\mathcal{F} \not\subseteq \mathcal{V}$ do
 8:     $l = \mathrm{MinHeapPop}()$;
 9:     for $f \in \mathcal{F}_l$ such that $r_f \geq s_l$ do
10:         $E = E \cup \{(l, f)\}$;
11:         if $f \notin \mathcal{V}$ then
12:             $r_f = s_l$;
13:             $\mathcal{V} = \mathcal{V} \cup \{f\}$;
14:             for $l' \in \mathcal{L}_f$ such that $r_f < s_{l'}$ do
15:                 $E = E \cup \{(f, l')\}$
16:                 $a_{l'} = a_{l'} - s_l$
17:                 $s_{l'} = a_l/|\mathcal{F}_l \setminus \mathcal{V}|$;
18:                 MinHeapUpdateKey(value = $l'$, newKey = $s_{l'}$);
19:             end for
20:         end if
21:     end for
22:     $\mathcal{V} = \mathcal{V} \cup \{l\}$;
23: end while
24: return $\langle \mathcal{G} = \langle V, E\rangle, \{s_l, \forall l \in \mathcal{L}\}, \{r_f, \forall f \in \mathcal{F}\}\rangle$;
```

D. Computing Link Gradients

This section describes two algorithms for computing derivatives in a network. Algorithm 3 calculates the derivative $\frac{\partial T}{\partial c_{l^*}}$ by perturbing the capacity of l^* by an infinitessimally small constant δ. We then measure the change produced in the total throughput, and divide by δ to calculate the rate of change. Since the bandwidth allocation function is piecewise linear, this slope is exactly the derivative $\frac{\partial T}{\partial c_{l^*}}$. While this method is accurate, it requires recomputing the rates r'_f from scratch, which is an expensive operation. Thus, we call this algorithm $BruteGrad$. We can improve the algorithm somewhat by replacing $ComputeBS$ in lines 1 and 3 with $FastComputeBS$. We call this improved algorithm $BruteGrad^{++}$. While asymptotically faster than $BruteGrad$, it is still slow if many derivatives need to be computed.

In contrast, Algorithm 4 ($ForwardGrad$) uses the information captured in the bottleneck structure graph itself to speed up the computation of the derivative. The key insight for

Algorithm 3 BruteGrad($\mathcal{N} = \langle \mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\}\rangle, l^* \in \mathcal{L}$)

```
1: $\langle \mathcal{G}, \{s_l\}, \{r_f\}\rangle = ComputeBS(\langle \mathcal{L}, \mathcal{F}, \{c_l\}\rangle)$
2: $c'_l = \begin{cases} c_l + \delta & l = l^* \\ c_l & \text{o.w.} \end{cases}, \quad \forall l \in \mathcal{L}$
3: $\langle \mathcal{G}', \{s'_l\}, \{r'_f\}\rangle = ComputeBS(\langle \mathcal{L}, \mathcal{F}, \{c'_l\}\rangle)$
4: return
       $\dfrac{\sum_{f \in \mathcal{F}} r'_f - r_f}{\delta}$
```

this algorithm is that once the bottleneck structure has been computed, it can be reused to calculate different derivatives without the need to recompute the bottleneck structure for each derivative, as in the $BruteGrad$ algorithm. The algorithm is inspired by forward mode automatic differentiation ("Forward Prop"), an algorithm for finding the derivative of a complicated expression that repeatedly applies the chain rule to larger and larger pieces of the expression [11]. In our case, the bottleneck structure is analogous to a computation graph of a complicated expression, since a flow's rate is determined by its bottleneck links, which in turn depend on its predecessors in the bottleneck structure. But the analogy fails in two ways. First a flow's rate can be affected by a change in its sibling's rate that frees up extra bandwidth in their shared parent, even if the parent's overall capacity stays the same. Second, a flow's rate can fail to change when its parent link changes, if it also has another parent bottleneck link that does not change. Thus, while the algorithm begins with the independent variable and propogates the derivatives forward according to the chain rule, it sometimes needs to backtrack in the graph to correct for these cases. Still, the algorithm is a significant improvement on $BruteGrad$. It only requires visiting each link or flow at most once, and it only visits nodes which are affected by changes in l^*. This means that $ForwardGrad$ has a much lower asymptotic complexity than $BruteGrad$. In the extreme case, l^* could have no descendants in the bottleneck structure, and the algorithm will terminate immediately.

Algorithm 4 ForwardGrad($\mathcal{N} = \langle \mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\}\rangle, l^* \in \mathcal{L}$)

```
 1: $d_s = 0 \quad \forall s \in \mathcal{L} \cup \mathcal{F}$
 2: $d_{l^*} = 1$
 3: MinHeapAdd(key $=\langle r_{l^*}, d_{l^*}/|\mathrm{children}(l^*, \mathcal{G})|\rangle$, value $=l^*$)
 4: $V = \emptyset$    # the set of previously visited nodes
 5: repeat
 6:     $l = \mathrm{MinHeapPop}()$
 7:     $V = V \cup \{l\}$
 8:     for $f \in [\mathrm{children}(l, \mathcal{G})]$ do
 9:         $d_f = d_l/|\mathrm{children}(l, \mathcal{G})|$
10:         for $l' \in \mathrm{children}(f, \mathcal{G}) \setminus V$ do
11:             $d_{l'} = d_{l'} - d_f$
12:             MinHeapAdd(key $=\langle r_{l'}, d_{l'}/|\mathrm{children}(l', \mathcal{G})|\rangle$, value $=l'$)
13:         end for
14:     end for
15: until MinHeapEmpty()
16: return $\sum_{f \in \mathcal{F}} d_f$
```

III. Benchmarks

A. Dataset and Experimental Environment

To ensure the benchmarks are performed on a realistic dataset, our team was given access to a set of anonymized

Algorithm 5 ForwardGrad($\mathcal{N} = \langle \mathcal{L}, \mathcal{F}, \{c_l, \forall l \in \mathcal{L}\} \rangle, f^* \in \mathcal{F}$)

```
1: $d_s = 0 \quad \forall s \in \mathcal{L} \cup \mathcal{F}$
2: $d_{f^*} = 1$
3: $V = \emptyset$    # the set of previously visited nodes
4: Let $b \in \mathcal{L}$ be a link such that $f^* \in$ children$(b, \mathcal{G})$    # b is the bottleneck link of $f^*$
5: Remove edge from $b$ to $f^*$ in $\mathcal{G}$
6: for $l \in$ children$(f^*, \mathcal{G}) \cup \{b\}$ do
7:     $d_l = d_l - d_{f^*}$
8:     MinHeapAdd(key $= \langle r_l, d_l/|\text{children}(l, \mathcal{G})|\rangle$, value $= l$)
9: end for
10: repeat
11:     $l = $ MinHeapPop()
12:     $V = V \cup \{l\}$
13:     for $f \in$ [children$(l, \mathcal{G})$] do
14:         $d_f = d_l/|\text{children}(l, \mathcal{G})|$
15:         for $l' \in$ children$(f, \mathcal{G}) \setminus V$ do
16:             $d_{l'} = d_{l'} - d_f$
17:             MinHeapAdd(key $= \langle r_{l'}, d_{l'}/|\text{children}(l', \mathcal{G})|\rangle$, value $= l'$)
18:         end for
19:     end for
20: until MinHeapEmpty()
21: return $\sum_{f \in \mathcal{F}} d_f$
```

NetFlow [12] logs from ESnet. ESnet is a high-performance network built to support scientific research that provides services to more than 50 research sites, including the entire US National Laboratory system, its supercomputing facilities, and its major scientific instruments [13].

The dataset contains NetFlow logs from February 1st, 2013, through February 7th, 2013. At the time the logs were generated, ESnet had a total of 28 routers and 78 links distributed across the US. (See Fig. 3 for a view of the ESnet topology at the time the logs were captured.) The dataset includes samples from all the routers, organized in intervals of 5 minutes, from 8am through 8pm, for a total of 1008 NetFlow logs for each router (or a total of 28224 logs across the network). The total data set is about 980 GB.

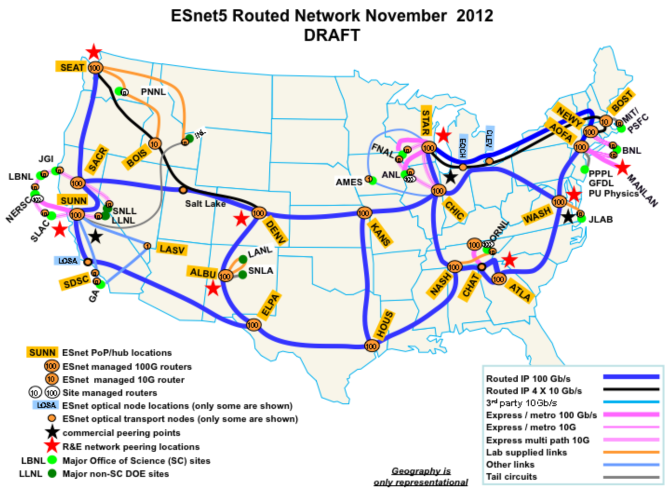


Fig. 3: ESnet network topology at the time the dataset we use in our benchmark was taken (February 2013). Source: ESnet historical network maps https://www.es.net/engineering-services/the-network/network-maps/historical-network-maps.

All tests were performed on an Intel Xeon E5-2683 v3 processor clocked at a rate of 2 GHz. The processor had 4 cores configured with hyperthreading disabled. L1, L2 and L3 caches had a size of 32 KB, 256 KB and 35840 KB, respectively, and the size of the RAM was 32 GB.

We benchmarked two software packages developed by our team for computing bottleneck structures. The first is a Python package that was previously published in [5]. This package implements the $ComputeBS$ algorithm for computing bottleneck structures and the $BruteGrad$ algorithm for computing link gradients. The second is a new C++ package equipped with a Python interface and functions to plot the bottleneck structure graph. It implements the $FastComputeBS$ algorithm for computing bottleneck structures and the $BruteGrad^{++}$ and the $ForwardGrad$ algorithms for calculating link gradients.

B. Computing Bottleneck Structures at Scale

In this section, we benchmark and compare the two programs on the task of computing bottleneck structures. We expect the C++ package to be more efficient because it is written in a faster language and uses an asymptotically faster algorithm.

1) Runtime: Figure 4 plots the time taken by each package to compute the bottleneck structure of ESnet at each of the 1008 logging snapshots. The seven separate days on which logs were collected are clearly distinguishable, corresponding to varying levels of traffic through the network (the gaps in our logs between 8 pm and 8 am each day are not represented in the plot). As expected, the C++ package is significantly faster than the Python package. The C++ package runs in 0.21 s on average, completing each in under 0.44 s, while the Python package averages 20.4 s and takes as long as 66.5 s. On average, the C++ package performs 87 times faster at this task.

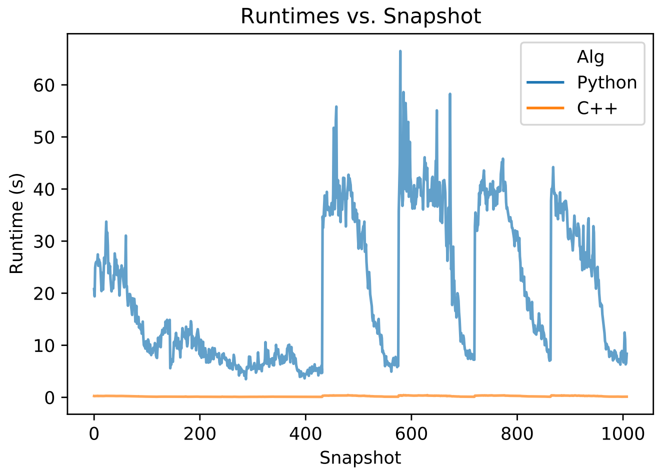


Fig. 4: Runtimes of the two packages on each of the 1008 network snapshots, showing that the new package runs quickly on each.

Figure 5 demonstrates the asymptotics of the $FastComputeBS$ algorithm. The left panel plots the observed run time of the C++ package against the asymptotic bound $|E|\log|\mathcal{L}|$, showing very high correlation between the two. This indicates that the asymptotic bound tightly captures the true running time of the algorithm. The right panel plots the runtime of each snapshot against the number of flows $|\mathcal{F}|$ present in the network at that time, also showing strong agreement. This is because, in our experiments, the number of links is the same across all snapshots, and since each flow traverses a small number of links, $|E|$ is approximately linear in $|\mathcal{F}|$.

2) Memory Usage: Figure 6 plots the amount of memory used by each package when computing the bottleneck structure of ESnet at each of the 1008 logging snapshots. Both algorithms must build a directed graph with the same numbers of vertices and edges. However, as Figure 6 shows, the C++ package is still far more efficient, using 26.7 MB on average. This represents a 4x median improvement over the Python package.

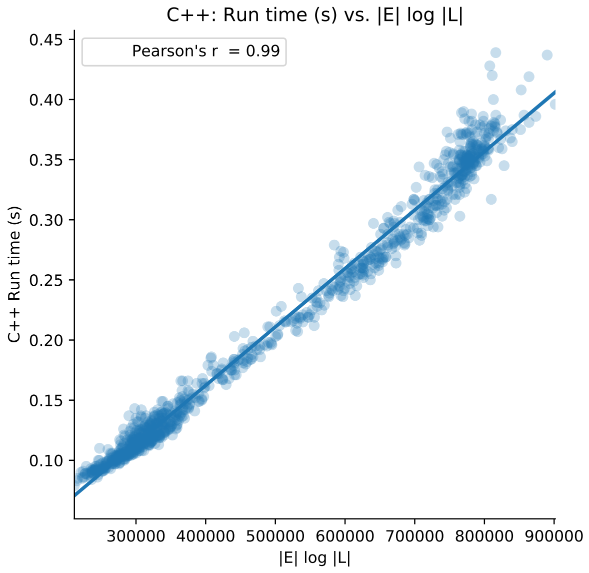
Figure 7 demonstrates the space complexity of the $FastComputeBS$ algorithm, showing that the amount of memory it uses is linear in the size of the input network.

C. Computing Link Gradients at Scale

In this section, we benchmark and compare the two programs' functionality for computing link gradients. We consider three methods in all: the Python package's $BruteGrad$, the C++ package's $BruteGrad^{++}$, and the C++ package's $ForwardGrad$. This allows us to separate the effect of using a faster algorithm from the effect of using a faster programming language. We consider one snapshot per hour over twelve hours. For each snapshot, we compute the derivative of the network's total throughput with respect to each of its links using each of the three algorithms.

1) Runtime: Figure 8 shows the runtime of each algorithm across all the links and snapshots on a log scale. The 12 different snapshots form discernible sections, since the state of the network remains constant throughout all trials within each snapshot. Changing from the Python package's $BruteGrad$ to the C++ package's $BruteGrad^{++}$ reduces the average runtime from 19.9 s to 0.30 s, a 66-fold improvement. Notice that this is approximately the same improvement observed when moving from Python's $ComputeBS$ to C++'s $FastComputeBS$, since these algorithms are used as subroutines by $BruteGrad$ and $BruteGrad^{++}$. Changing to the C++ package's $ForwardGrad$ algorithm further reduces the runtime to 0.09 s, a further 3.5-fold improvement. This level of performance makes it possible to compute a large number of derivatives in real time to respond to rapidly changing network conditions.

As discussed in Section II-D, when $ForwardGrad$ is used to compute a link derivative, the runtime is linear in the number of flows and links that are affected by the given link. This group, which we call the link's "region of influence", is simply the descendants of the link in the bottleneck structure graph. [2] In contrast, the run times of the $BruteGrad$ and $BruteGrad^{++}$ algorithms depend on the size of the entire network, since they reconstruct the whole bottleneck structure.



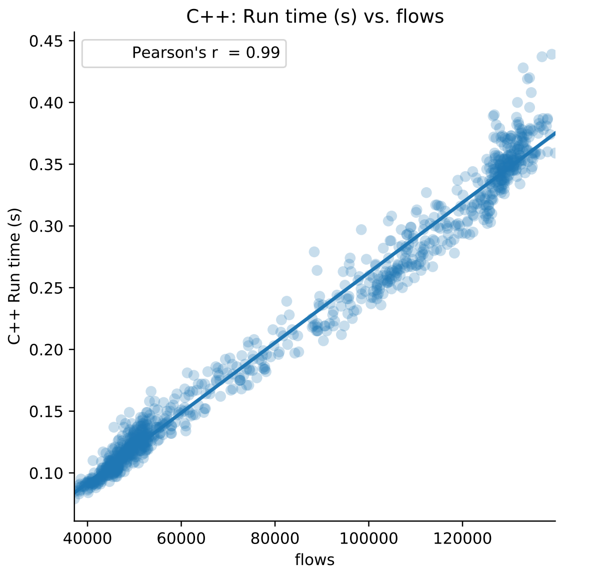


Fig. 5: Asymptotics of the $FastComputeBS$ algorithm, showing that it is quasilinear in the size of the network.

Figure 9 plots the runtimes of the three algorithms against the size of the given link's region of influence and against the total number of flows in the network. As expected,

[2]In rare cases, a single flow may be bottlenecked simultaneously at multiple links. In this case, the siblings of a link's descendants may also be part of the region of influence, even if they are not themselves descendants of the given link. We observe no such cases in our experiments.

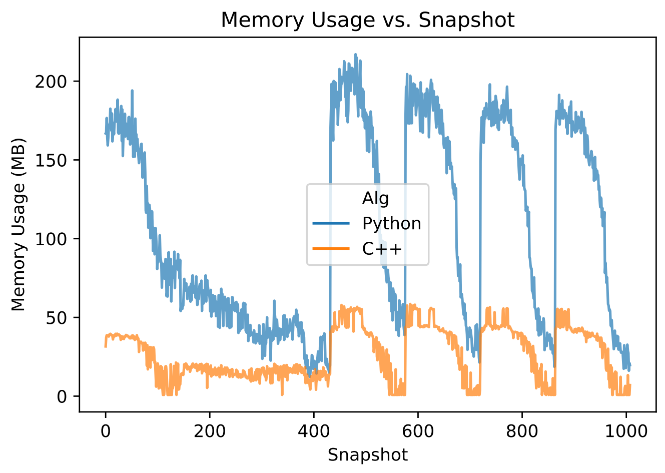


Fig. 6: Memory usage of the two packages on each of the 1008 network snapshots, showing that the new package uses about 4x less space.

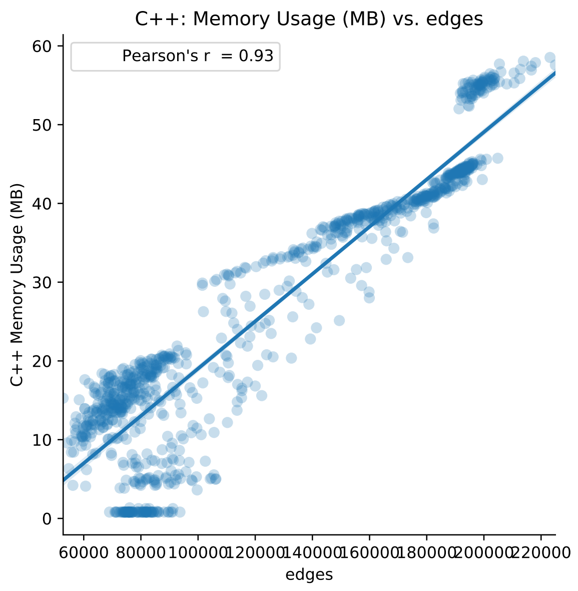


Fig. 7: Asymptotics of the space complexities of $FastComputeBS$, showing that it is linear in the size of the input network.

$ForwardGrad$ is highly correlated with the former (top left). It is also somewhat correlated with the number of flows (top right), but only because networks with many flows also tend to have some links with many descendants. Even in these large networks however, the runtime falls under the line of best fit for most links. As the middle and bottom left panels show, the runtimes of $BruteGrad^{++}$ and $BruteGrad$ are not well explained by the size of the region of influence. Instead, like $FastComputeBS$ and $ComputeBS$, they are linearly

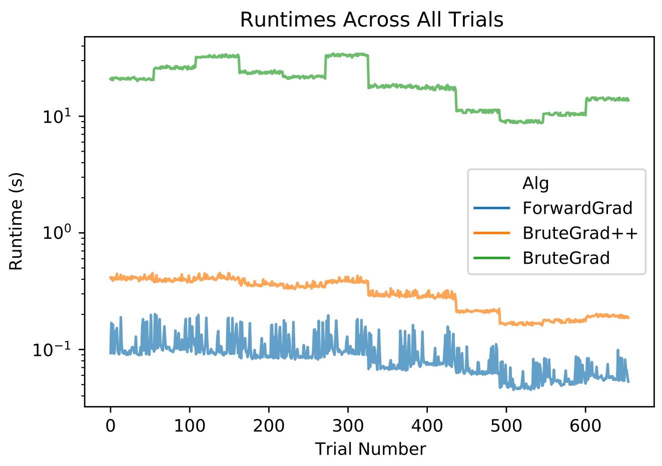


Fig. 8: Runtimes of algorithms for computing link derivatives across 655 trials from 12 snapshots of the network.

dependent on the size of the network (middle and bottom right panels).

Given their time complexities, we expect $ForwardGrad$ will exhibit a larger speed-up compared to $BruteGrad^{++}$ in cases when the input link has a small region of influence. Figure 10 plots this relationship, showing that the speed-up factor grows as the size of the region of influence approaches 0. This is because the size of region of influence shrinks in comparison to the network as a whole. Thus, the 3.5x average speed-up observed in our experiments would keep increasing as the algorithms are applied to larger and larger networks.

2) Memory Usage: We profile the algorithms based on the amount of additional memory they need to compute each derivative given a pre-constructed bottleneck structure. Figure 11 shows that replacing the Python package's $BruteGrad$ with $BruteGrad^{++}$ significantly reduces the memory usage—by a factor of 10 on average. Replacing $BruteGrad^{++}$ with $ForwardGrad$ has an even greater impact, reducing memory usage by a factor of 30 on average. Indeed, the average amount of additional memory used by $ForwardGrad$ across all trials was just 850 KB, and the maximum was 6.4 MB. (The steep decline in memory usage observed in the later trials reflects the fact that the number of flows in the network decreased precipitously at the end of the day.)

Figure 12a shows the asymptotic behavior of the $ForwardGrad$' memory usage. Unlike the other algorithms, $ForwardGrad$ does not use more memory as the network size increases. Technically, the space-complexity of $ForwardGrad$ is linear in the size of the region of influence, since forward grad stores a derivative value for each element in that set. In our experiments however, we find that this dependence is so weak as to make the memory usage almost constant. (See Figure 12b. If we only consider trials in the middle 99% by memory usage, to exclude outliers, then the correlation shrinks to 0.06.) These experiments demonstrate that the $ForwardGrad$ algorithm is highly scalable and

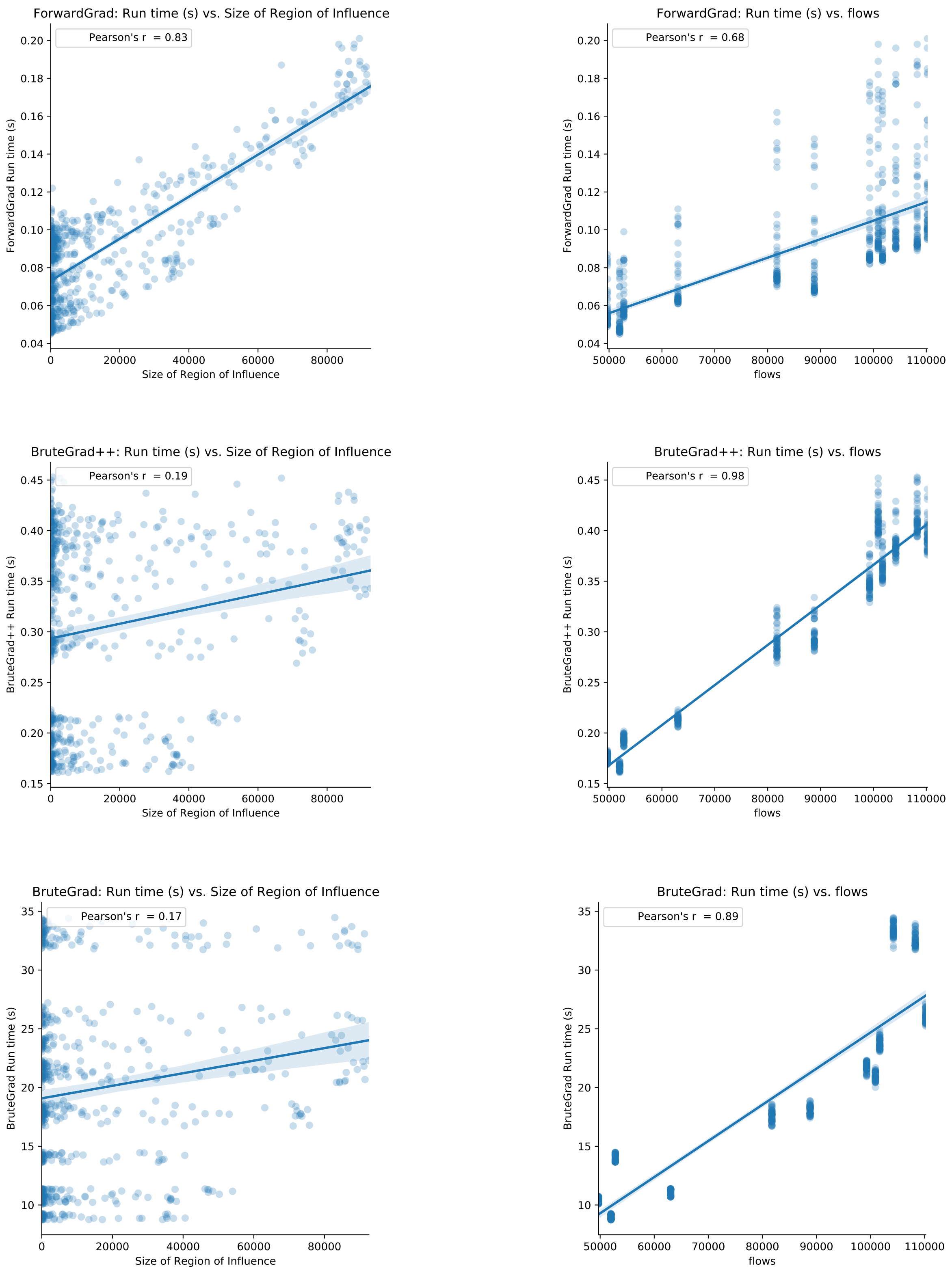


Fig. 9: Asymptotic behavior of the algorithms to compute bottleneck structures. $ForwardGrad$'s runtime is linear in the size of the region of influence, while $BruteGrad$ and $BruteGrad^{++}$ grow with the size of the network as a whole.

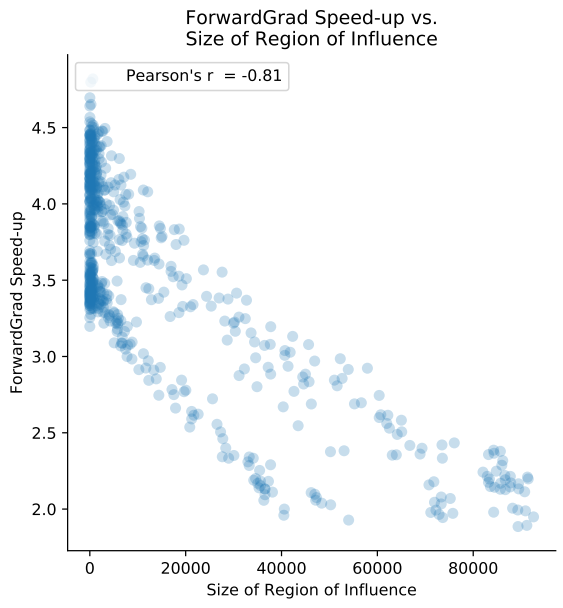


Fig. 10: The speed-up factor obtained by replacing *BruteGrad*$^{++}$ with *ForwardGrad* increases as the region of influence shrinks.

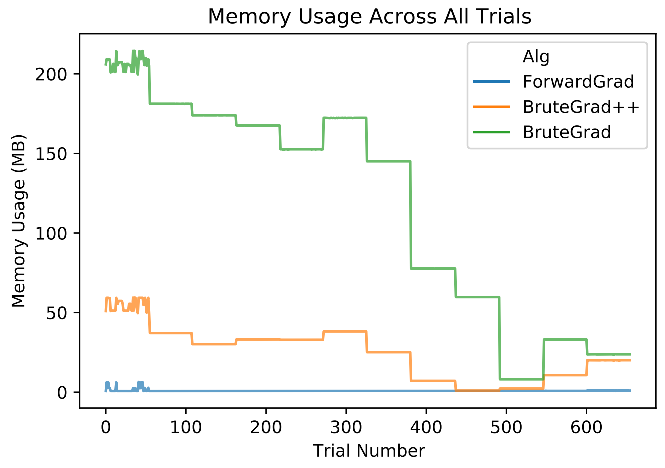
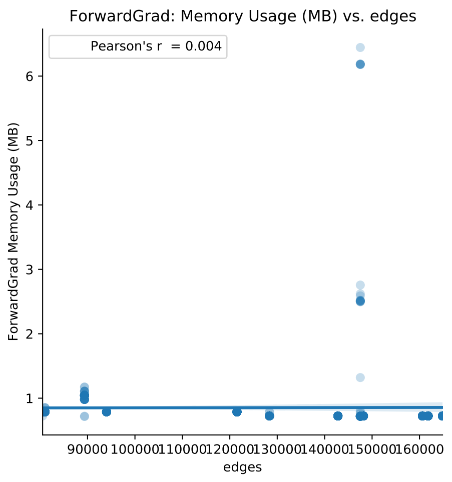
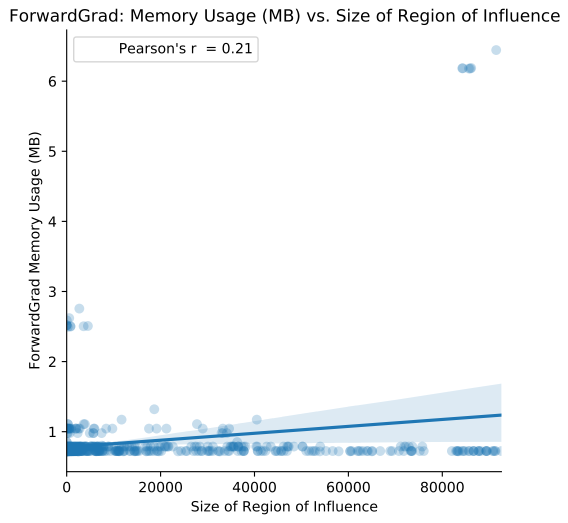


Fig. 11: Memory usage of each algorithm across all trials. *ForwardGrad* is orders of magnitude more space-efficient.



(a) *ForwardGrad*'s memory usage does not grow with the size of the network.



(b) *ForwardGrad*'s space complexity is linear in the size of the region of influence, but actual memory usage is nearly constant.

Fig. 12: Memory usage of *ForwardGrad*.

space-efficient.

IV. Related Work

The analysis of bottlenecks in data networks has been the subject of intense research since 1988, when Van Jacobson proposed the first congestion control algorithm. This advance literally saved the Internet from congestion collapse [14]. However much of the research during the past three decades has been premised on the notion that a flow's performance is uniquely determined by the capacity of its bottleneck and the communication round trip time of its path. This view has lead to dozens of congestion-control algorithms based on characterizing (whether implicitly or explicitly) the performance of each flow's bottleneck. Well-known works in this vein include BBR [3], Cubic [2] and Reno [1]. While these algorithms have been crucial to the success of large-scale communication networks like the Internet, they continue to treat bottlenecks as independent elements and do not consider their interactions or dynamic nature.

One line of research has taken a more global view by modeling networks as instances of multi-commodity flow problems. The classical formulation of these problems is altered to include a notion of fairness between competing flows

[15]. This approach has been applied to routing and load balancing problems under the assumption of multi-path routing; algorithms typically involve iteratively solving a series linear programs and adjusting the constraints [16]. This approach has a high computational complexity that makes scaling difficult [17], despite algorithmic tricks to mitigate the cost [18]. Moreover, this framework is somewhat brittle; it obscures the roles played by individual elements in determining network behavior, lacking, for example, an equivalent notion to link and flow derivatives.

The existence of such complex interactions among bottlenecks has not gone completely unnoticed in the research community. For instance, [19] states that ”the situation is more complicated when multiple links are involved; [...] As flows are added or deleted and advertised fair-share rates are adjusted, bottleneck links for flows may change, which may in turn affect other bottleneck links, and so on, potentially weaving through all links in the network.” However, the authors do not attempt to solve the problem of modeling these complex relationships.

The solution to this problem was first presented in [4]. This work introduced the concept of latent bottleneck structures and used a directed graph to model them. It also introduced the first algorithm to compute the bottleneck structure, which appears in this paper as $ComputeBS$. However, no benchmark was provided, leaving open the question of whether such bottleneck structures can be computed efficiently enough to be used in real production networks.

The first software package for computing bottleneck structures and using them to analyze networks was introduced in [5]. The authors provide Python implementations of the $ComputeBS$ and $BruteGrad$ algorithms, along with functionality for reading sFlow logs and performing simulations. We use their package as a baseline in this paper. However, the performance of this software was not benchmarked, and the core functionality was too slow and memory intensive for use with large networks in practice.

This paper provides the first benchmark of the proposed algorithms to compute bottleneck structures, demonstrating that, when efficiently implemented, they are capable of scaling to support the size of real production networks. This result confirms the practical usefulness of bottleneck structures as a framework to help network operators understand and improve performance with high-precision.

V. Using $FastComputeBS$ and $ForwardGrad$ in Production Networks

The algorithms described in this paper are developed as part of the GradientGraph (G2) technology [5]. G2 is a network optimization software package that leverages the analytical power of bottleneck structures to enable high-precision bottleneck and flow performance analysis. Network operators can use G2 to address a variety of network optimization problems, including traffic engineering, congestion control, routing, capacity planning, network design, and resiliency analysis, among others.

The G2 technology is composed of three layers: the core analytical layer, the user interface (northbound API) and the network interface (southbound API).

The core analytical layer constructs the bottleneck structure of the network under study using $FastComputeBS$ and uses algorithms such as $ForwardGrad$ (among others from the Theory of Bottleneck Structures [4], [7]) to analyze performance. Then, G2 provides network operators with both online and offline recommendations on how to configure the network to achieve better performance. Online recommendations address traffic engineering problems and include actions such as changing the route of a set of flows or traffic shaping certain flows to improve overall system performance. Offline recommendations address capacity planning and network design problems and include actions such as picking the optimal link to upgrade or identifying the most cost-effective allocation of link capacities (for instance, identifying optimal bandwidth tapering configurations in data center networks [20]).

The user interface (northbound API) provides three mechanisms to interact with G2's core analytical engine: a representational state transfer (REST) API to enable interactive and automated queries, a graphical user interface (GUI) that allows operators to visualize bottleneck structures and gradients, and a command line interface (CLI).

The network interface (southbound API) provides a set of plugins that allow for convenient integration of G2 into production networks. These plugins read logs from flow monitoring protocols such as NetFlow [12], sFlow [21] or SNMP [22]. The sets of links $\mathcal{L}$ and active flows in the network $\mathcal{F}$ can be easily reconstructed if such a monitoring protocol is enabled in all the routers and switches of the network. Otherwise, links and flows can be reconstructed with additional information extracted from SNMP (to learn the network topology) and from routing tables (to infer flow path information). The capacity parameters $\{c_l, \forall l \in \mathcal{L}\}$ can be obtained from SNMP or static network topology files that production networks typically maintain. G2's southbound API includes plugins for all of these standard protocols to enable its integration with production networks.

VI. Conclusion

This paper presents the first practical application of the Theory of Bottleneck Structures to production networks. In a series of experiments on the ESnet network, we show that our new software package far outperforms the one published in [5] on the core operations of computing bottleneck structure graphs and computing link gradients. We also show that our $FastComputeBS$ and $ForwardGrad$ algorithms are highly scalable in both time and space complexity. $FastCompute$ is shown to scale quasilinearly with the size of the network, and $ForwardGrad$ is shown to scale linearly with the size of the region of influence. These results demonstrate that bottleneck structure analysis is a practical tool for analyzing production networks. The benchmarks indicate that our package can analyze networks that are even larger than ESnet and do so in real time, even as network conditions are changing rapidly. The efficiency of our core algorithms enables them to be used as subroutines in larger network optimization toolchains. The advances presented in this paper unlock the potential of bottleneck structure analysis for myriad important applications.

Acknowledgment

We thank the ESnet team at the Lawrence Berkeley National Laboratory for providing the anonymized NetFlow dataset used in the presented benchmarks and for their support throughout this process.

This work is supported by the Department of Energy under contract number DE-SC0019523.

References

[1] K. Fall and S. Floyd, "Simulation-based Comparisons of Tahoe, Reno and SACK TCP," *SIGCOMM Computer Communication Review*, vol. 26, no. 3, pp. 5–21, July 1996. [Online]. Available: http://doi.acm.org/10.1145/235160.235162

[2] S. Ha, I. Rhee, and L. Xu, "CUBIC: A New TCP-friendly High-speed TCP Variant," *SIGOPS operating systems review*, vol. 42, no. 5, pp. 64–74, July 2008. [Online]. Available: http://doi.acm.org/10.1145/1400097.1400105

[3] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control," *ACM Queue*, vol. 14, no. 5, pp. 50:20–50:53, October 2016. [Online]. Available: http://doi.acm.org/10.1145/3012426.3022184

[4] J. Ros-Giralt, A. Bohara, S. Yellamraju, M. H. Langston, R. Lethin, Y. Jiang, L. Tassiulas, J. Li, Y. Tan, and M. Veeraraghavan, "On the bottleneck structure of congestion-controlled networks," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 3, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3366707

[5] J. Ros-Giralt, S. Yellamraju, A. Bohara, R. Lethin, J. Li, Y. Lin, Y. Tan, M. Veeraraghavan, Y. Jiang, and L. Tassiulas, "G2: A network optimization framework for high-precision analysis of bottleneck and flow performance," in *2019 IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)*, 2019, pp. 48–60.

[6] D. M. Friedlen and M. Z. Nashed, "A note on one-sided directional derivatives," *Mathematics Magazine*, vol. 41, no. 3, pp. 147–150, 1968. [Online]. Available: http://www.jstor.org/stable/2688187

[7] (2020) Technical report on the theory of bottleneck structures and its applications. Reservoir Labs. Available upon request: contact@reservoir.com. New York, NY, USA.

[8] (2019) Mininet-extensions-anonymized: Mininet extensions to support the analysis of the bottleneck structure of networks. [url]. [Online]. Available: https://github.com/reservoirlabs/g2-mininet

[9] E. Martelli and S. Stancu, "Lhcopn and lhcone: status and future evolution," in *Journal of Physics: Conference Series*, vol. 664, no. 5. IOP Publishing, 2015, p. 052025.

[10] R. Rao, "Synchrotrons face a data deluge," *Physics Today*, 2020. [Online]. Available: https://physicstoday.scitation.org/do/10.1063/PT.6.2.20200925a/full/

[11] R. D. Neidinger, "Introduction to automatic differentiation and matlab object-oriented programming," *SIAM Rev.*, vol. 52, no. 3, p. 545–563, Aug. 2010. [Online]. Available: https://doi.org/10.1137/080743627

[12] B. Claise, G. Sadasivan, V. Valluri, and M. Djernaes. (2004) Netflow specifications, cisco systems. [Online]. Available: https://www.ietf.org/rfc/rfc3954.txt

[13] ESnet, "ESnet Energy Sciences Network," 2019. [Online]. Available: http://es.net/network-r-and-d/experimental-network-testbeds/test-circuit-service/

[14] V. Jacobson, "Congestion Avoidance and Control," *SIGCOMM computer communication review*, vol. 18, no. 4, pp. 314–329, August 1988. [Online]. Available: http://doi.acm.org/10.1145/52325.52356

[15] M. Allalouf and Y. Shavitt, "Maximum flow routing with weighted max-min fairness," in *QofIS*, 2004.

[16] D. Nace and M. Pioro, "Max-min fairness and its applications to routing and load-balancing in communication networks: a tutorial," *IEEE Communications Surveys Tutorials*, vol. 10, no. 4, pp. 5–17, 2008.

[17] E. Danna, A. Hassidim, H. Kaplan, A. Kumar, Y. Mansour, D. Raz, and M. Segalov, "Upward max-min fairness," *J. ACM*, vol. 64, no. 1, Mar. 2017. [Online]. Available: https://doi.org/10.1145/3011282

[18] D. Nace and L. Doan, "A polynomial approach to the fair multi-flow problem," *Rapport Interne, Heudiasyc, UTC*, 2002.

[19] L. Jose, L. Yan, M. Alizadeh, G. Varghese, N. McKeown, and S. Katti, "High Speed Networks Need Proactive Congestion Control," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XIV. New York, NY, USA: ACM, 2015, pp. 14:1–14:7. [Online]. Available: http://doi.acm.org/10.1145/2834050.2834096

[20] G. Michelogiannakis, Y. Shen, M. Y. Teh, X. Meng, B. Aivazi, T. Groves, J. Shalf, M. Glick, M. Ghobadi, L. Dennison, and K. Bergman, "Bandwidth steering in hpc using silicon nanophotonics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356145

[21] P. Phaal, S. Panchen, and N. McKee, "sFlow Specifications, InMon Corporation," *IETF RFC 3176*, 2001.

[22] J. Case, M. Fedor, M. Schoffstall, and J. Davin. (1990) A simple network management protocol (snmp). [Online]. Available: https://tools.ietf.org/html/rfc1157