`tpprof`: A Network Traffic Pattern Profiler

Nofel Yaseen
University of Pennsylvania
nyaseen@seas.upenn.edu

John Sonchack
University of Pennsylvania
jsonch@seas.upenn.edu

Vincent Liu
University of Pennsylvania
liuv@seas.upenn.edu

Abstract

When designing, understanding, or optimizing a computer network, it is often useful to identify and rank common patterns in its usage over time. Often referred to as a network traffic pattern, identifying the patterns in which the network spends most of its time can help ease network operators' tasks considerably. Despite this, extracting traffic patterns from a network is, unfortunately, a difficult and highly manual process.

In this paper, we introduce `tpprof`, a profiler for network traffic patterns. `tpprof` is built around two novel abstractions: (1) network states, which capture an approximate snapshot of network link utilization and (2) traffic pattern subsequences,which represent a finite-state automaton over a sequence of network states. Around these abstractions, we introduce novel techniques to extract these abstractions, a robust tool to analyze them, and a system for alerting operators of their presence in a running network.

1 Introduction

When designing, understanding, or optimizing a computer network, it is often useful to identify common patterns in its usage over time. Often referred to as a **network traffic pattern**, identifying the patterns in which the network spends most of its time can result in useful insights:

- *All-to-all traffic*, which might manifest as uniform utilization of all paths between a set of application nodes, might suggest the importance of bisection bandwidth and guide future provisioning decisions.
- *Chronic stragglers*, where we expect all-to-all traffic but a significant amount of time is spent with only a few flows active, might suggest the need for better sharding and mitigation techniques.
- *Elephant flow dominance*, in which utilization is dominated by isolated path-level hotspots, might guide future provisioning decisions.
- Finally, *synchronized requests/responses*, indicated by repeated bursts of cross-network communication all originating at a single node, might motivate changes in the application or network architecture.

While there are a number of existing tools that capture flow- and switch-level trends (e.g., heavy hitter analysis [68], network tomography [28], or the vast array of network analytics suites on the market [1–6, 30]), identifying prevalent *network*-level patterns typically requires a significant amount of manual effort and specialized analyses. To determine the presence of synchronized requests/responses, for instance, an operator might need to instrument the start and stop times of all flows in the system, correct for the time drift of different machines, compute the cluster tendencies of the data (e.g., with a Hopkins statistic or heuristic), and distinguish it from all-to-all traffic by examining the sources and destinations of synchronized flows. To determine whether this pattern is a particularly common one would require additional analyses.

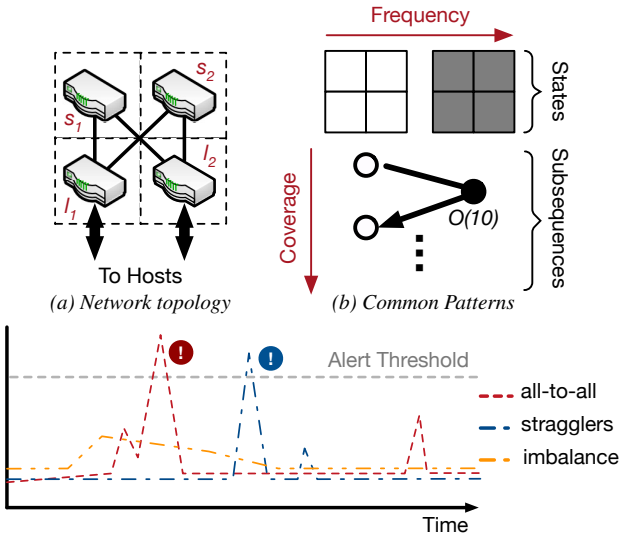


Figure 1: `tpprof`'s visualizations for b common traffic patterns and c the TPS score over time for a simple leaf-spine topology, a. We describe these in more detail later, but in b, states are heatmaps of common utilization patterns over the network in a; darker is hotter. Subsequences are common transition patterns between the aforementioned states. These are ranked by their frequency of occurrence and their cumulative coverage of the profiled run, respectively. The subsequence shows an all-to-all pattern: the network starts unutilized (left state), becomes fully utilized (right state) for 10s of samples, then returns. In c, `tpprof` is tracking three different known traffic patterns. When the score of any of them crosses the alerting threshold (twice in the figure), `tpprof` deduces that the pattern has occurred in the network.

Our goal in this work is a tool for the automatic identification of the most prevalent traffic patterns in a network. To that end, we present the design and implementation of `tpprof`, a network traffic pattern profiler.

Similar to traditional application profilers like gprof [27] or Oprofile [22] that have helped programmers understand and improve their software for decades, `tpprof` automatically measures, extracts, and ranks common traffic patterns of individual applications within running networks. It also facilitates the monitoring of known patterns so that, when specific patterns appear in the network, the operator is informed. It does both of these things without modifying applications and without affecting existing network traffic—the only changes we require are to switch monitoring configurations. Examples of both of the above tools in action are shown in Figure 1.

Traffic patterns are, unfortunately, significantly more challenging to profile than applications. Traditional profilers benefit from well-defined building blocks (functions or lines of code) connected by well-defined call graphs. In contrast, networks offer little such structure: switch and link utilizations are noisy and measured in real values (Bps); their evolution over time is even less constrained. In the end, two different instances of something as simple as all-to-all traffic will never look exactly the same.

Thus, `tpprof` is built around two novel abstractions: (1) *network states*, which capture an approximate snapshot of a network's device-level utilization and (2) *traffic pattern subsequences*, which represent a finite-state automaton over a sequence of network states. As hinted above, subsequences serve as both output and input to our system: output in the case of profiling an existing network, input in the case of specifying a traffic pattern alert. In both cases, classification of network states and sub-sequences is approximate and implemented through specialized clustering techniques.

We implement and deploy `tpprof` to a small hardware testbed in order to monitor and profile the traffic patterns of real distributed applications like memcache, Hadoop, Spark, Giraph, and TensorFlow. We demonstrate that, using `tpprof`, we can find meaningful patterns and issues in their behavior. Further, we demonstrate `tpprof`'s utility on larger and more complex networks by profiling a trace taken from one of Facebook's frontend clusters. While our evaluation focuses on data center networks (where interesting and impactful distributed applications are plentiful), `tpprof` and its techniques generalize to arbitrary networks.

Specifically, this paper makes the following contributions:

- **Novel abstractions for describing common traffic patterns:** We introduce two abstractions, network states and traffic pattern subsequences, that together enable network operators to easily describe and reason about common traffic patterns. Network states capture similar configurations of approximate utilization of a specific application or set of applications running in a network. Subsequences are then strings of states with bounded repetition that summarize traffic pattern changes over time.
- **Domain-specific algorithms for clustering and ranking both network states and subsequences:** Through empirical analysis of a variety of application traffic patterns, we identify and design algorithms that transform a network trace into the building blocks of traffic patterns. Specifically, we demonstrate through PCA and waypoint analysis of real application traffic that GMMs are well-suited to capturing first-order similarities between different network utilization patterns. In the case of subsequences, we create a domain-specific clustering algorithm that extracts sequences that are both common and that provide broad coverage of the measured network traces.
- **A language and mechanism for expressing and fuzzily matching known traffic patterns in observed traces:** Finally, to complement the above, we develop a simple grammar for describing traffic patterns and introduce an algorithm that automatically identifies approximate occurrences of known traffic patterns within network traces. Our scoring engine outputs a confidence score that can be used to generate alerts when known traffic patterns appear in observed traces.

Taken together, `tpprof` is, to the best of our knowledge, the first profiling tool for network-wide traffic patterns. Our implementation is in Python and the code is open source.[1]

2 The Anatomy of a Traffic Pattern

We begin by introducing the definitions and abstractions on which `tpprof` is built. First and foremost, we define the overall traffic pattern of a network as follows:

Network Traffic Pattern — A function $f(x,t)$ that represents, for an entire network N across a time span $[t_0,t_1]$, the utilization of device $x \in N$ at time $t_0 \leq t \leq t_1$.

We also define, for each application in the network:

Application-specific Traffic Pattern — $f_A(x,t)$, equivalent to the network traffic pattern, but only accounting for a single application or set of applications, A.

For the purposes of our clustering and ranking algorithms, the distinction is unimportant; unless otherwise specified, we use 'traffic pattern' to refer to both. Instead, the choice of whether to filter by application is entirely the user's (with the mechanisms in Section 4); Regardless, for a given network and overall workload, we note that the traffic pattern of both the network and its constituent applications will typically exhibit predictable and repeated characteristics given a sufficiently long measurement period. These patterns can occur over short time spans of individual packets and flows, or over longer time spans in the form of diurnal or weekday/weekend effects.

A contribution of this paper is the decomposition of traffic patterns into a more convenient low-level primitive:

[1] https://github.com/eniac/tpprof.

Network Sample — $|N|$ real values that capture an approximate snapshot of $f(x,t)$ for all devices $x \in N$, at a particular time t, and averaged over the last t_Δ seconds.

Network Sample Sequence — A chain of network samples that sample $f(x,t)$ over increments of t_Δ, where t_Δ is bounded by the measurement granularity of the system.

Any traffic pattern can be described in these terms. For the network in Figure 1a, the all-to-all pattern in Figure 1b is one example. Another is chronic stragglers, which we can describe as a transition between two configurations, assuming a load balanced network: (1) all switches at high utilization and (2) only l_1, l_2 and s_1 at high utilization; or the same but replacing s_1 with s_2.

We can perform a similar exercise for all of the many (possibly application-specific) traffic patterns in the literature, e.g., rack-level hotspots in data centers [24, 29, 44, 58], synchronized behavior of distributed applications [9, 20, 67], and stragglers in data-intensive applications [21, 41, 43]. We do the same for the link- and switch-level traffic patterns that are the focus of most existing automated profiling tools [1–6, 30].

While not necessarily the way these patterns were described originally, sequences of network samples provide a general primitive with which we can represent arbitrary patterns.

3 `tpprof` Design Overview

Our goal in this paper is the design and implementation of a *profiler* for network and application-specific traffic patterns. Our system, `tpprof`, is intended to identify traffic patterns, rank them in prevalence, and assist network operators in monitoring for their recurrence. Like other profiling tools, `tpprof` is not intended to improve networks directly; rather, its focus is on assisting users with designing, understanding, and optimizing them.

On that note, we take inspiration from traditional sampling profilers like gprof [27], Oprofile [22], and Valgrind [48]. These profilers take an unmodified application and they periodically sample system state (e.g., stack traces) to produce a statistical profile of the target application. Early instantiations solely sampled program counters; over time, they expanded to capture trends in function utilization and call graph traversal.

`tpprof` uses a similar approach to construct profiles of traffic patterns. To that end, network samples and sequences of samples present an attractive substrate. In principle, a sequence of network samples creates a statistical profile of an application's network utilization. Unfortunately, these samples are unlikely to ever repeat: small differences in application processing time, workload, and background traffic can cause substantive differences in traffic, as can slight noise in the sampling frequency of the measurement framework. Extracting patterns from raw samples is challenging.

Core abstractions. To address this challenge, we introduce two additional abstractions:

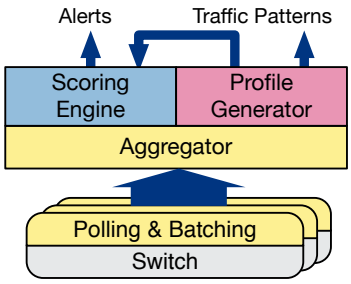


Figure 2: The overall architecture of `tpprof`. `tpprof` polls, batches, and aggregates switch counters from the network. These are fed into (1) a scoring engine that alerts on detection of known patterns and (2) a profile generator that extracts common traffic patterns from the gathered trace.

Network State — A class of network samples defined by a single, n-device network sample, S, and n variance values, $\bar{v}$ such that S is a centroid of the multivariate normal distribution with shape defined by $\bar{v}$.

Network State Subsequences — A class of sequences of network states that allows for bounded repetition of states. A state subsequence can be represented as a regular expression or finite state automata of network states.

Multiple network samples can be mapped to a single network state and multiple sample sequences can be mapped to a single state subsequence. These abstractions are tolerant to noise by design: variations of link utilization from sample to sample are smoothed by our method of extracting network states; variations in the evolution of those samples over time are smoothed by our method of extracting subsequences. The precise construction of both of the above elements is described in Sections 5.1 and 5.2.

Components. `tpprof` consists of three primary components:

1. A configurable *sampling framework* that periodically samples the device-level utilization of a specified application, set of applications, or the full network (Section 4).
2. A *profiling tool* for the automatic extraction and visualization of the most common states and state subsequences in the captured data (Section 5).
3. An *alerting system* that scores incoming traces against a set of user-defined patterns using a fuzzy string search in order to facilitate network automation (Section 6).

Of the above, only (1) affects the network itself; (2) and (3) occur out-of-band. As such, the overhead of `tpprof` is minimal: in the case of an non-application-specific traffic pattern, little is required beyond an SNMP poller; application-specific patterns only require simple `iptables` and switch configuration changes on top of that.

Our current implementation leverages programmable switches and a recently proposed network-wide monitoring framework [63]. This provides slightly more control and accuracy than an implementation based on top of traditional

switches, but it is not a strict requirement; we detail both approaches in Section 4.

Workflow. tpprof profiles production networks. A typical workflow thus proceeds as follows. First, users specify three configuration parameters: the start time a, the end time b, and the sampling interval i. The network can optionally be configured to track certain applications separately. Regardless, a centralized service periodically polls the byte counters of the entire network between time a and b, with interval i.

The centralized service will stream the data through a set of scoring algorithms that quantify the prevalence of a set of target patterns in the measured trace. If the score of the trace exceeds a threshold for a given pattern, an alert will be generated. By default, the measurement data is not stored. This changes when users request a profile, i.e., a visualization, of common traffic patterns in the network. In this case, raw network samples are stored for a specified profiling duration for clustering and analysis. The resulting profile can be used to construct additional pattern alerts or analyzed separately. The remainder of this paper describes each of the three components of tpprof in more detail.

4 Sampling Framework

tpprof's sampling framework continually polls a custom set of switch counters to capture traffic patterns. Most production networks already implement some form of this—tpprof can piggyback on these existing polling suites. tpprof is, however, parameterized by at least two configuration options.

- *Application filters:* To profile application-specific traffic patterns, users must provide a proper filter for the traffic in question. In tpprof, this takes the form of iptables rules. Any filter that can be expressed as an iptables rule is allowed. Thus, multiple applications can be captured by a single filter and different flows from the same application can be split into different filters by port, packet type, etc. All traffic matching installed filters are marked with a special set of bits, e.g., in the DSCP field of the packet header. We term the value of these bits a *filterid*.
- *Sampling interval:* Users must also specify an interval, t_Δ, at which tpprof's sampling framework will poll all devices in the network. This interval is common to the entire system, so the network and all application-specific traffic patterns will be read at this rate. Though this is a user-defined value, we anticipate that it should be set to the minimum value feasible for the target network without incurring sample loss. We note that, because the raw data is discarded after alert pattern matching, measurement data storage capacity is not a bottleneck in tpprof.

4.1 Counter Implementation and Sampling

Network devices in a tpprof-enabled network track a set of device-level application-specific byte counters corresponding to the space of possible *filterid*s. For every packet traversing the switch, the counter associated with the specified *filterid* is incremented by the size of the packet; all categories summed will give the cumulative byte counter of the device. In this design, the network is never reconfigured; instead, users associate applications to *filterid*s directly through the iptables rules at every end host.

tpprof samples these counters at an interval of t_Δ via a recently proposed measurement primitive, Speedlight. For brevity, we omit the details of its operation and refer interested readers to its non-channel-state variant [62, 63]. At a high level, the primitive is that of a synchronized, causally consistent snapshot of network-wide switch counters. Compared to SNMP and other naïve polling tools, Speedlight provides increased accuracy and low minimum sampling interval, both of which are useful when profiling network traffic patterns.

Alternative implementations. We note that, at its core, the only requirement of tpprof is for configurable counters and a method to periodically poll all such counters in the network. There are other implementations that satisfy this requirement.

For instance, most modern switches typically include support for configurable ACL entries with per-entry counters. This approach has the advantage that it can be implemented without end host cooperation. Class of Service (CoS) counters are similarly promising. Note that, if application-specific tracking is not required, periodic SNMP polling is sufficient.

4.2 Batching and Aggregation

While it is possible to directly transmit polled counter results to a centralized profiling service, the scale of measurement data collected by tpprof necessitates careful handling. In particular, there are two issues we must address: decreasing overhead and handling sample loss.

For the first, to decrease the number of messages and the overhead per sample, tpprof agents running on each network device assemble results locally before shipping batches of size B in the following format:

```
sampleBatch: {
  switch: <SWITCH_ID>,
  indexes: [i : <SAMPLE_ID> for i from 0 to B],
  app1_bytes: [i : <BYTE_COUNT> for i from 0 to B],
  ...
  appM_bytes: [i : <BYTE_COUNT> for i from 0 to B]}
```

indexes[k] and *_bytes[k] should correspond to a single network sample. Gaps in the samples, e.g., from failures or measurement packet drops, will manifest as gaps in the indexes array. In these cases, tpprof attempts to interpolate values by taking the difference between byte counters before and after any gap and averaging the difference over the length of the gap. If the device has rebooted or if it stays down for too long, we will treat the device as 'failed' during the missing measurement intervals. 'Failed' devices are excluded from

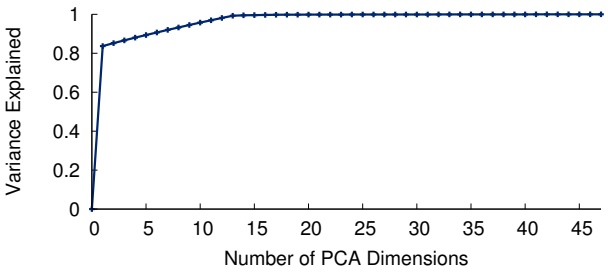


Figure 3: Covariance explained by different numbers of PCA dimensions. Dataset is a trace of utilization over 48 ToR switches in a Facebook frontend cluster.

profiling and treated as wildcards during alerting. Note that reboots are also excluded from interpolation as we do not know how much traffic was sent before the counter was reset.

Storing data for profiling. While `tpprof` does not store raw counter values in the common case, raw values are necessary for generating profiles. Profiles are, therefore, executed on-demand using the API:

```
start_profile(start, end, filter_id)
```

The duration of collection should be long enough to capture a representative slice of behavior. In general, longer is better, but this may be subject to limitations of sample storage space and the user's timeline. `filter_id` $= -1$ indicates the sum of all application-specific counters.

5 The `tpprof` Profiling Tool

We first discuss how `tpprof` extracts and ranks traffic patterns before delving into the scoring and alerting system in Section 6. To that end, the output of the previous subsection (4.2) is a network sample sequence, i.e., a sequence of n-device samples of network utilization. Using that, the output of the `tpprof` profiler is a ranked list of network states and a ranked list of state subsequences, as sketched by Figure 1b and demonstrated in Section 7. `tpprof` achieves this using a pair of domain-specific clustering techniques that are designed to capture first-order patterns in network traffic.

5.1 Network States

The first challenge in identifying meaningful traffic patterns is the inherent noise present in a trace of network samples. Small variations in workload, TCP effects, background traffic, or any number of other factors mean that, most likely, no two network samples will look exactly alike.

To de-noise the data, `tpprof` summarizes network samples into a small number of distinct network states. We can naturally frame this as a clustering problem, where the points to be clustered are the n-element vectors representing network samples. Clustering has been used to great effect in a number of fields, from image segmentation to recommendation systems and anomaly detection; each of these has its own set of challenges and associated clustering algorithms.

Network state extraction is no different in that regard. In this work, we leverage empirical analysis of a variety of applications and traces to identify and design algorithms suited to the domain. Applications observed include Hadoop, Giraph, TensorFlow, Spark, Memcache, and a trace from a production Facebook frontend cluster (see Section 7 for their details).

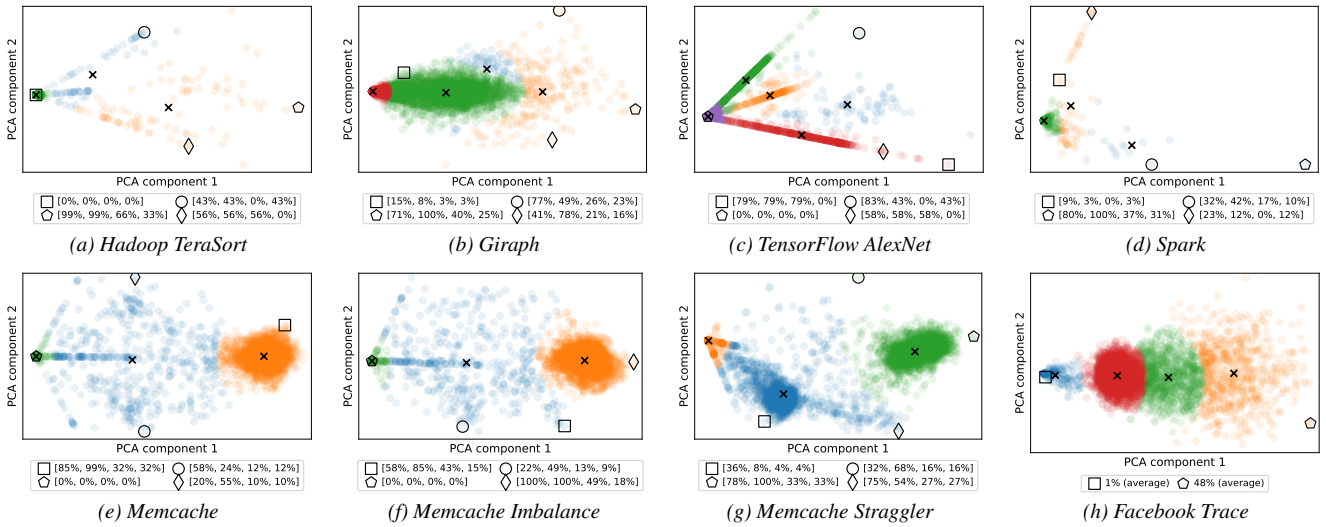
Dimensionality reduction. Before delving into `tpprof`'s clustering algorithm, we note that, in general, networks present a particular challenge to clustering because of their high device counts. Profiling the ToRs of a 48-rack data center cluster, for instance, might result in a 48-feature input vector, which prior work has indicated might be too many dimensions for typical distance metrics [17].

The general solution to this well known 'curse of dimensionality' [15] is transforming the data into a lower-dimensional space before clustering. The simplest approach is to cluster on only a subset of features. While this works in other domains, it is not well suited for our problem because the load on *every* device may be important. Instead, `tpprof` preprocesses data with Principle Component Analysis (PCA) [25], which derives a small set of features that are an orthogonal linear transformation of the original features. Said differently, PCA removes redundancies in the original data by creating a new set of independent features that explain most of the variability in the original data.

PCA is most effective when features are strongly correlated, and there is good reason to believe that this is true in our domain. Recent work [21] shows that network usage is highly correlated, driven by the data-parallelism in distributed systems [31, 65]. Analysis of the Facebook traffic trace verifies this: each ToR had strong and statistically significant correlations ($r > .7$, $p < .001$) with an average of 3.25 other ToRs. The applications we profiled showed similar results.

Figure 3 measures the effect of PCA on that data, gauged by plotting the number of PCA dimensions (i.e., features) versus explained variance. All other traces we obtained showed similar results. A value of 1 means that a PCA transformation to K dimensions preserved all the information contained in the original data with 48 dimensions. Even for this large and complex trace, one dimension already explains over 80% of the variance and two dimensions explain $\sim$85%. Striking a balance between clustering efficacy and explained variance, `tpprof` projects all data into 2D by default. This can be adjusted depending on the data.

Gaussian Mixture Models (GMMs) for sample classification. `tpprof` clusters around typical network variations through its use of GMMs. To demonstrate this effect, we consider the 2D PCA projections described above and visualized, for our set of profiled applications and traces, in Figure 4. To help interpret points in the PCA space, we also plot network load at 4 extreme *waypoints* along the convex hull of each trace. We observe two general cluster shapes in the projected



(a) Hadoop TeraSort *(b) Giraph* *(c) TensorFlow AlexNet* *(d) Spark*

(e) Memcache *(f) Memcache Imbalance* *(g) Memcache Straggler* *(h) Facebook Trace*

Figure 4: Network samples projected into a 2-dimensional PCA space. Cluster centers are marked with x's. Shaped-markers map points in the space to sample vectors [l1, l2, s1, s2] (see Figure 1a) or, for the Facebook trace, average utilization.

data: 'rays' and 'clouds'.

- **Rays**, like the ones prominent in Figures 4a and 4c, are typically associated with rising or falling utilization on a set of highly correlated nodes. We can see this effect most clearly in Figure 4c through the relationship between ⬠, ♢, and □. Compare their utilizations with that of ○.
- **Clouds**, like the ones in Figures 4b and 4g, typically characterize samples that are similar in configuration, but separated by noise that offsets points by a small amount in all directions of the PCA space. These clouds can be more or less dense, depending on the coherence of the pattern. The memcache variants, for instance, exhibit strong all-to-all behavior, which manifests as dense clouds to the right of the PCA plots.

Synchronized behavior and noise around a specific configuration capture most of the key behavior in our empirical tests. For these two types of clusters, GMMs are known to perform well. GMMs model a cluster as a multivariate Gaussian with independent parameters for each dimension of the data. This independence provides the flexibility for clusters to fit both types of clusters with arbitrary densities. We fit GMMs to the data using the expectation-maximization algorithm from Scikit-learn [51], which finds clusters that are each defined by a centroid sample and a vector of per-feature variances.

Automated detection of cluster count. GMMs are defined in terms of a fixed number of clusters, K. `tpprof` selects K automatically by using a Bayesian Information Criteria (BIC) score. Informally, a better (lower) BIC score means that a specific clustering, if used as a generative function, is more likely to produce the observed data.

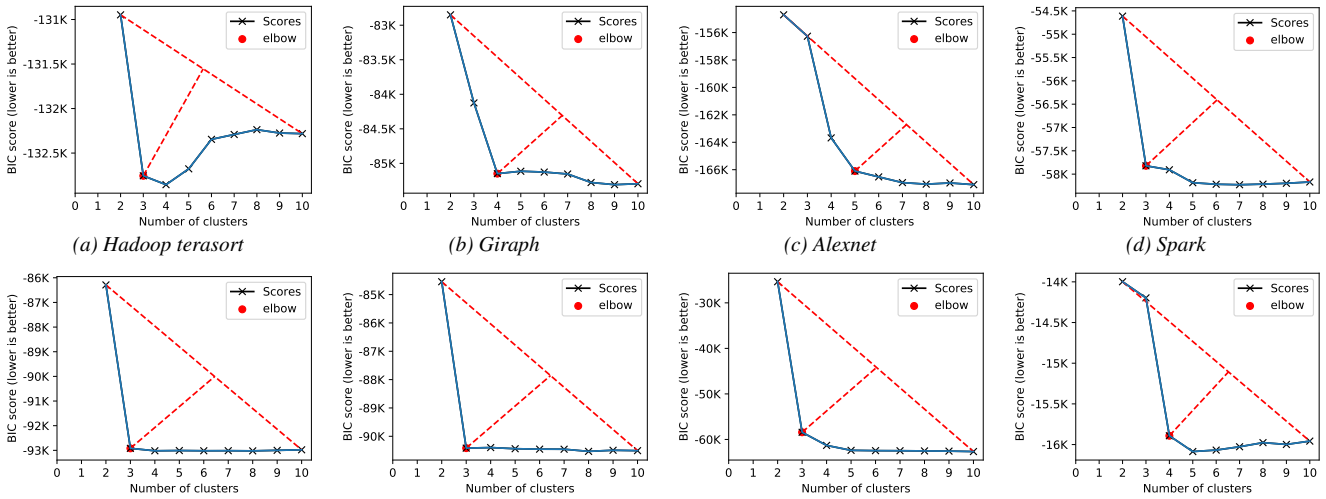
We note, however, that BIC scores tend to improve as K increases, but a high number of clusters can overfit the data. To overcome this issue, we select a K value at which the benefit gained by adding an extra cluster starts to diminish. Finding such "elbows", or points of maximum curvature, is a common problem in machine learning and systems research. We use the Kneedle method [56], a simple but general algorithm based on the intuition that the point of maximum curvature in a convex and decreasing curve is its local minima when rotated θ degrees counter-clockwise about (xmin, ymin) through the line formed by the points (xmin, ymin) and (xmax, ymax). Specifically, we plot the *BIC* score versus K and draw a line segment connecting the points for $K = 2$ and a configured maximum of $K = 10$, which we set based on the typical working set capacity of humans. The optimal value of K is given by the point furthest from that line. Figure 5 shows the results of this analysis for the applications and traces introduced above.

5.2 Network State Subsequences

Network state subsequences extend network states to capture temporal patterns in traffic. Like states, subsequences require compression of the full sequence of samples taken during the profiling run into a small set of representative patterns. Unlike states, existing sequence-based clustering algorithms are a poor fit for network traffic patterns.

To see why identifying and ranking network state subsequences is challenging, consider a strawman solution: take all possible subsequences of the trace and count their frequencies, e.g., the trace ABC would result in the following subsequences $\{A \times 1, B \times 1, C \times 1, AB \times 1, BC \times 1, ABC \times 1\}$.

Challenge 1: (a) $A^5 = AAAAA$ versus (b) $A^5B \ldots AAB \ldots AAB$
Intuitively, the interesting bit of sequence (a) is that there is a long run of A's. The strawman solution will instead output that the most common subsequence and frequency is the single state $A \times 5$, followed by $AA \times 4$, etc. With the naïve approach, short subsequences will always take



(a) Hadoop terasort *(b) Giraph* *(c) Alexnet* *(d) Spark*

(e) Memcache *(f) Memcache imbalance* *(g) Memcache straggler* *(h) Facebook cluster*

Figure 5: Selecting the number of clusters with Bayesian Information Criteria (BIC) and the elbow heuristic.

param *stateSequence[# samples in the trace]*: Full sequence of network states.
param *minFreq*: The minimum number of subsequence occurrences before it is counted as a 'common' subsequence.

```
1  Function getSubSequences:
2      for targetLength : len(stateSequence) to 2 do
3          maxStart ← len(stateSequence) − targetLength
4          for start : 0 to maxStart do
5              end ← start + targetLength
               /* Skip taken ranges */
6              if [start,end] contained in takenRanges then
7                  continue
               /* Add if it meets minFreq */
8              subseq ← log10Merge(stateSequence[start:end])
9              if (# subseq observations) ≥ minFreq then
10                 Add (start, end) to takenRanges after loop
11                 Increment subseqs[subseq]
12             else
13                 Hold subseq until the threshold is reached
14     subsequenceCoverage ← computeCoverage(subseqs)
15     totalCoverage ← computeTotalCoverage(subseqs)
16     return subseqs, subsequenceCoverage, and totalCoverage
```

Figure 6: Pseudocode for finding common subsequences in a sequence of network states.

priority; in fact, we can prove that subsequences will *never* beat their member states. On the other hand, sequence (b) demonstrates a case where it might be useful to be able to observe the shorter subsequences. In this case, greedily setting aside the A^5 would miss the third occurrence of AAB, which is arguably the more important pattern.

Challenge 2: $XA^{39}Y \ldots XA^{40}Y \ldots XA^{41}Y$
The strawman solution also performs poorly with similar, but not identical ranges. While it may find here that there are long strings of As, or even that X is typically followed by As, or that Y is typically preceded by As, it will fail to find that As are typically sandwiched between X and Y. Variance in duration is common in networks, where measurement timing, available capacity, and workload size changes frequently.

Challenge 3: $(AB)^3(CDEFGHIJKLMNOPQRSTUVXYZ)^2$
Finally, we note that frequency itself is not an ideal metric. Consider the above trace. The longer trace is much rarer and more interesting, but a pure frequency analysis will rank AB higher in importance.

`tpprof`'s subsequence extraction (outlined in Figure 6) addresses these challenges through a series of rules, which we describe below. Line numbers reference Figure 6.

Only consider subsequences of length 2+ [Line 2]. While knowing the most frequent single states is useful, the goal of extraction is to capture patterns in traffic. We, therefore, prune subsequences of length 1 from consideration and list the relative frequency of single states separately.

Ignore strict subsequences [Lines 6–7]. To better summarize cases like Challenge 1(a), we exclude any subsequence that is wholly contained within another subsequence. We implement this efficiently using two data structures: (1) *takenRanges*, a list of existing subsequences sorted by *start*, and (2) a *heap*-based index of the currently overlapping subsequences, sorted by *end* (not shown).

Frequency threshold before a subsequence is counted [Lines 9–13]. The above rule, applied directly, might produce a single subsequence encompassing the entire trace. To account for this, we set a minimum frequency threshold, *minFreq*, before which the subsequence is not counted. We note that a lower value of *minFreq* promotes the inclusion of longer but sparser subsequences, while a higher value favors many, shorter subsequences. `tpprof` automatically tunes this value using the `hyperopt` library to optimize for 'total coverage', a metric we describe at the end of this section.

Log10 repetition frequencies [Line 8]. To handle cases like

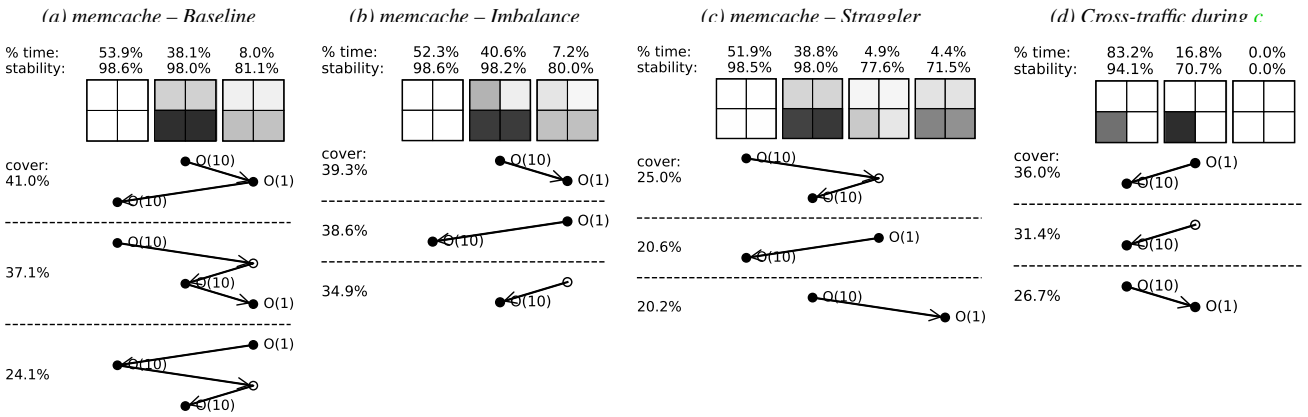


Figure 7: `tpprof` profiles of memcache in three different environments (a–c), plus a profile of cross-traffic (d) active during c.

that of Challenge 2, common in network traces, we compress repetitive states into the nearest power of 10. Doing so ignores small differences in duration while still retaining the length's rough magnitude.

Coverage rather than frequency [Lines 14–15]. As evidenced in Challenge 3, differences in the length of subsequences and the ability of subsequences to overlap diminish the utility of frequency as a way to reason about the relative importance of different subsequences. Instead, we propose *coverage* as a metric for ranking subsequences and for hyperparameter-tuning minFreq. Coverage measures, for either a single subsequence or the union of all subsequences, the cumulative fraction of states in the stateSequence that are included in at least one subsequence.

We encourage the reader to step through several short examples of network state sequences to see why the above rules produce intuitive results.

5.3 Example Visualization: memcached

To tie the above discussion together and showcase the utility of `tpprof`, we present to the reader several real profiles produced by the `tpprof` tool suite. See Section 7 for a description of the hardware testbed used in these tests.

As a baseline, we first look at a memcache workload generated using the `memaslap` [7] benchmarking utility, running in isolation. Each machine in the testbed was configured as a memcache server with 64 B keys and 1024 B values. Gets and sets were randomly generated from two machines—one in each rack—with a ratio of 9:1. In this simple test, the two memcache clients, every 6 s, will simultaneously begin performing 290k get/set operations.We profiled this behavior, collecting a total of 7000 network samples at a 50 ms interval.

Visualization structure. Figure 7a shows the `tpprof` profile for this run. Like Figure 1b, heatmaps of *network state* are at the top of the figure and the most common *network state subsequences* are below. Each heatmap shows the centroid of the sample clusters it represents. We add to this the state's *% time* (the amount of time the network spends in the state) as well as its *stability* (the probability that the network, once in the state, will stay there); states are sorted by % time.

For subsequences, we include the top three by coverage; more can be generated on demand. Subsequences are depicted with a series of points (representing states) connected by arrows (denoting transitions between states). The points align horizontally with the states they represent. Solid points accompanied with an O(x) label indicate an x–10x repetition of that state. The number on the left of each subsequence is the percentage of the trace that it covers. Note that coverage can add to more than 100% due to overlapping subsequences.

Observations. We can observe several characteristics in Figure 7a. First, we can see that there are three states in which the network spends its time. In the one that accounts for more than half the trace, the network is unutilized. The other two show different states of even leaf and even spine utilization, indicating that the network is relatively balanced when it is being used. Note that the leaves of the network are consistently hotter than the spines due to rack-internal communication.

As expected of the workload, the subsequences of the profile show a trace composed of *on-off* periods of *all-to-all* traffic. We can also deduce from the duration of repetitions that the on and off periods both last on the order of seconds. Further, we can infer that the network takes time to ramp up/down from full utilization. This is inferred from the presence of the (L-to-R) 3rd state and the absence of direct transitions between states 1 and 2. Ramp ups seem to be an order of magnitude faster than ramp downs.

`tpprof`'s observations can inform network and application changes. For example, if an operator were to see a similar profile in practice, she could conclude that load balancing is not an issue. Instead, a more promising approach would be to either desynchronize traffic to spread out utilization over time or augment the leaf switches with additional capacity.

5.3.1 Case Study #1: Detecting Load Imbalance

`tpprof` can also help to detect acute problems in networks. As a case study, we artificially introduced an ECMP miscon-

⟨*signature*⟩ ::= **{ (**⟨*target state set*⟩**) ;** ⟨*target sequence*⟩ **}**

⟨*target state set*⟩ ::= ⟨*target state*⟩ **,** ⟨*target state set*⟩

⟨*target state*⟩ ::= $\vec{utilization}$

⟨*target sequence (P)*⟩ ::= ⟨*target state*⟩ | ~⟨*P*⟩
| ⟨*P*⟩∧⟨*P*⟩ | ⟨*P*⟩∨⟨*P*⟩
| ⟨*P*⟩* | ⟨*P*⟩**{** min repetitions **,** max repetitions**}**

Figure 8: Definition of a traffic pattern signature.

figuration [69] into the network. Specifically, we configured one of the ToR switches to only use the left spine; otherwise, the workload is identical to Figure 7a. Figure 7b shows the output of our `tpprof`'s Python-based visualizer. An operator comparing this profile to that of the baseline would be able to see the new and stark differences between the two spines in all states with load and conclude that ECMP was not doing its job. While imbalance can also be due to elephant flows and hash collisions, the fact that this happens consistently and always with the same spine points to a structural issue.

5.3.2 Case Study #2: Debugging a Noisy Neighbor

As another case study, we use `tpprof` to debug an apparent straggler in the system. In this experiment, we add a heavy background flow between two hosts connected to the lower-left leaf, l_1. Figure 7c shows the profile in question. From this profile, an operator can observe that, in 5–10% of samples, there is a slight bias toward l_1 while the other leaf is largely un-utilized. These samples are summarized in the right two network states. If the operator is expecting an even all-to-all pattern like the one in Figure 7a, these states would lead her to suspect that a task in the system is straggling.

`tpprof`'s ability to profile concurrent applications independently can also help to diagnose this issue. In particular, she can view the profile of non-memcache traffic present during the same profiling period. In this case, `tpprof` would provide her with Figure 7d, which clearly shows a competing flow or set of flows within l_1.

6 Traffic Pattern Scoring

The `tpprof` components described in prior sections allow users to profile their networks and find prominent traffic patterns. In many cases, after finding certain patterns, users are likely to want to know if (or when) they occur in the future. The `tpprof` traffic pattern scoring engine solves this problem. The key challenge is designing both a language that makes it simple for users to specify pattern signatures and also an algorithm efficient enough to detect those patterns in realtime.

Traffic pattern signatures. A traffic pattern signature describes the approximate spatial and temporal characteristics of a traffic pattern. It is defined by the grammar in Figure 8 and has two components.

- A set of *target network states* describe the approximate samples that are likely to be observed during the traffic pattern. These can be generated from prior profiling runs or manually specified.
- A *target subsequence*, written as a regular expression, that estimates how the network transitions between the target states during the pattern.

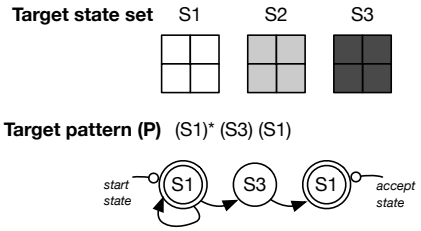


Figure 9: An example traffic pattern signature that detects a synchronized all-to-all burst.

As an example, Figure 9 illustrates a signature to detect a synchronized all-to-all burst of traffic in our example topology. The target states in the signature are: S_1, 0% utilization on all links; S_2, 50% utilization on all links; and S_3, 100% utilization on all links. The signature's target subsequence is, thus, one in which the network is in S_1 before transitioning to S_3 (i.e., high, all-to-all utilization) and immediately going back to S_1, signaling a quick end to the all-to-all utilization.

Scoring signatures. `tpprof`'s Traffic Pattern Score (TPS) algorithm quantifies a signature's prominence in a network sample sequence by finding and scoring subsequences that are similar to it. This amounts to a streaming fuzzy string search. Figure 10 illustrates the scoring algorithm for the all-to-all signature in Figure 9, while Figure 11 provides psuedocode of our streaming implementation. There are three steps.

1. *State matching:* The TPS algorithm first maps each incoming sample to the most similar target state, transforming the stream of samples into an *intermediate stream*.
2. *Pattern matching:* It then scans the intermediate stream for the target subsequence using a finite automata [34]. A match occurs when the automaton reaches an accept state, at which point it is executed in reverse to identify the start point of the longest matching subsequence.
3. *Match scoring:* A match indicates that the exact target subsequence has been found in the intermediate stream; however, how this relates to the underlying sample stream is unclear. Thus, the final step is to score match strength by calculating the average similarity between the two streams during the subsequence.

Writing signatures. There are two sources for signatures. First, they can be automatically generated by the profiler, from the network state subsequences it identifies. This allows the TPS algorithm to automatically identify future reoccurrences of events identified with the `tpprof` profiler.

Second, users can manually write signatures that characterize the most important attributes of a traffic pattern. Since TPSes use a fuzzy algorithm, patterns do not need to be exact. Instead, they can be defined programmatically. With the

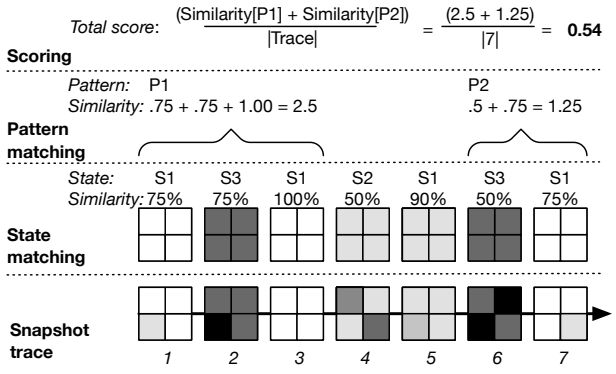


Figure 10: Matching and scoring a sample trace against the all-all signature in Figure 9.

```
1  signature ← (targetStates, regexp)
2  Function TPSGrep(signature, sampleStream):
3      Initialize matchStream
4      compile_patterns(matchStream)
5      scoreBuf ←[]
6      offset ← 0
7      for each (sample, timestamp) in trafficPattern do
8          /* Identify most similar target state.*/
9          stateSymbol ← nearestNeighbor(sample,targetStates)
10         similarity ← |netState − sample|
11         /* Track scores for up to BUF_LIM of the last samples. */
12         scoreBuf.append(score)
13         if len(scoreBuf)>BUF_LIM then
14             scoreBuf.pop()
15             offset ← offset + 1
16         /*Invoke HyperScan to update stream. */
17         (begin, end) = scan(matchStream, stateSymbol)
18         /*If a match occurred, calculate and emit a score. */
19         if end ≠ NULL then
20             emit sum(scoreBuf[begin-offset:end-offset]
```

Figure 11: The streaming TPS algorithm.

three primitives described below, users can express simple but powerful signatures.

- *State definition*, e.g., `(x:v, y:u)`, which defines a state with switch `x` having utilization `v` and switch `y` having utilization `u`.
- *Set assignment*, e.g., `X:v`. This sets every switch `x` ∈ `X` to utilization value `v`.
- *Iteration (over sets or switches)* e.g., `{(x:v) for x ∈ X}`, which defines a set of states: one state for each switch in `X`, defining that switch to utilization value `v`.

Table 1 lists five example signatures written with these primitives. We evaluate them later in Section 7.3.

7 Implementation and Evaluation

`tpprof` is implemented in Python/C++ as a standalone service that aggregates samples, profiles them, and scores them for the presence of known traffic patterns, as described in the previous sections. Each of the profiles shown in this section is a real output of `tpprof`, generated programmatically

Pattern	Signature	State Definitions
Short all-all	$S_1^* S_2 \{1,10\} S_1$	$\{S_1\}$=`N:0.0`,$\{S_s\}$=N`:0.5`
Long all-all	$S_1^* S_2 \{10,100\} S_1$	$\{S_1\}$=`N:0.0`,$\{S_s\}$=N`:0.5`
Hotspots	$(S_1 \mid S_2 \mid S_3 \mid S_4)\{10,100\}$	$\{S_1, ..., S_4\}$=`{(x:1.0, -x:0.0) for x ∈ N}`
Imbalance	$S_1^* \mid S_2^*$	$\{S_1, S_2\}$=`{(x:1.0, -x:0.0) for x ∈ (`s_1`, `s_2`)}`
Stragglers	$(S_1 \mid S_2 \mid S_3)^* S_3$	$\{S_1, S_2, S_3\}$=`{(`l_1`:v, -`l_1`:0.0) for v ∈ (0.1,0.01,0.0)}`

Table 1: Traffic pattern signatures for a leaf-spine network N with spines (s_1, s_2) and leaves (l_1, l_2).

using Python and Matplotlib 3.1.1. The requisite counters and polling/batching components that run on each device are implemented in P4 and Python, respectively. Traffic Pattern Scoring is implemented in C++ using `hyperscan` [34].

Hardware testbed. To verify the utility of `tpprof` and its outputs, we used it to profile and score the traffic patterns of real applications running on a small hardware testbed consisting of a Barefoot Wedge100BF-32X programmable switch connected to six servers with Intel(R) Xeon(R) Silver 4110 CPUs via 25 GbE links. The testbed is configured to emulate a small leaf-spine cluster like the one in Figure 1a. To implement this network, we split the Wedge100BF switch into 4 fully isolated logical switches. Each logical switch runs ECMP to balance load across paths.

Application workloads. On our hardware tested, we profile four popular networked applications, in addition to the memcache evaluation in Section 5.3:

1. Hadoop running a TeraSort [11] benchmark workload with 5B rows of data. Our Hadoop instance ran version 2.9.0 with YARN [12] on 10 mappers and 8 reducers spread across the 5 servers (and 1 master).
2. Spark's GraphX [13] running a connected components benchmark workload with 1.24M vertices. We ran Spark 2.2.1 with Yarn on 5 servers (and 1 master).
3. Giraph [10] running a PageRank synthetic benchmark workload with 120,000 vertices and 3,000 edges on each vertex. We used 23 workers in total across our 6 servers.
4. TensorFlow running the AlexNet [38] image processing model with 1 server managing parameters and 5 workers. We used ILSVRC 2012 data for training.

Unless otherwise specified, these applications were run in the presence of background TCP traffic derived from a well-known trace of a large cluster running data mining jobs [8]. Profiles are of the target application only.

Large-scale trace. To augment our small testbed, we also profile packet traces of 48 Top-of-Rack switches from three of Facebook's production clusters: a frontend cluster, a database cluster, and a Hadoop cluster. As the datasets are sampled by a factor of 30,000, we divide the timestamps by 30,000

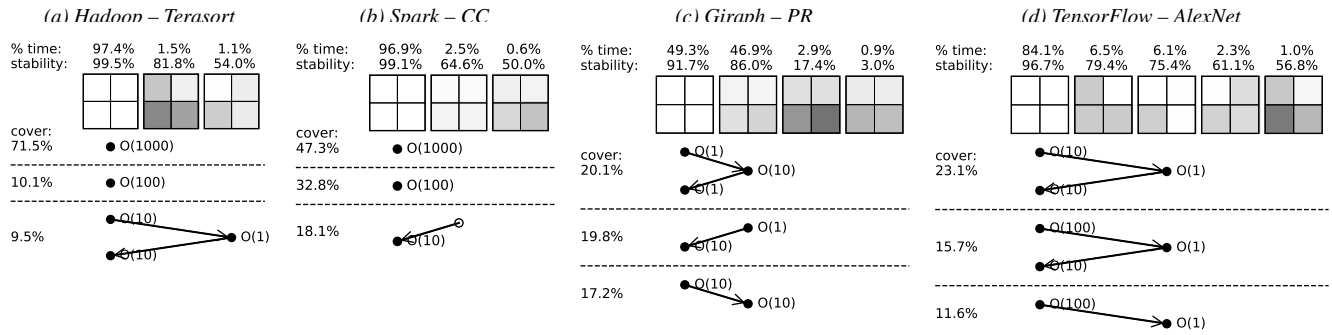


Figure 12: Profiles of more complex applications running with realistic background traffic.

to obtain an approximate representation of a full trace. Note that multiplying traffic by 30,000 would have given a more accurate distribution, but resulted in artificially stable patterns.

7.1 Profiling More Complex Applications

To evaluate how `tpprof`'s algorithms deal with more complex applications, we profile each of the application workloads we introduced earlier in this section. These applications all ran in the presence of background traffic, but we only show profiles of the application-specific traffic.

From the resulting profiles in Figure 12, we can see that, for the most part, the network was only lightly utilized during these tests. In Hadoop and Spark, for instance, the network spent $> 96\%$ of the time unutilized, indicating that our particular testbed tends to be CPU-bound. Giraph is the notable exception, spending about equal time utilized and not.

The states reveal some interesting behavior of the applications. For Hadoop and TensorFlow, we see heavy skew in spine utilization, but not to a consistent spine. This likely indicates the presence of a few large flows that dominate the network and sidestep ECMP's flow-level balancing. We also see in these two workloads a slight bias toward the lower-left switch. This is due to task placement: for Hadoop, that switch is home to the controller and name server; for TensorFlow, it holds both the chief worker and the parameter server.

7.2 Profiling Large Production Networks

`tpprof` is able to profile more complex networks as well. To demonstrate this, we run `tpprof`'s profiler over large-scale traces of the combined traffic for three production Facebook clusters and show the output in Figure 13. We separate the states and subsequences for readability.

Figure 13a shows the profile for the frontend cluster. As in the original paper describing this trace (Figure 5 of [54]), we can observe a clear split between the average utilization of cache, multifeed, and web servers. States A–C show memcache at full utilization, webservers at low utilization, and varying levels of multifeed traffic. Diverging from the original paper, we find an additional network state (occurring 3.8% of the time) in which the multifeed server utilizations spike. The stability of this state indicates that this may manifest as small, but intense and correlated bursts. Subsequences further show frequent transitions between states A and B, with state C representing a short-lived relative lull in multifeed traffic.

Figure 13b and Figure 13c show the profiles of a database and Hadoop cluster, respectively. Notably, the database cluster is very uniform and stable across the trace, indicating a steady workload and good load balancing properties. The Hadoop profile is also notable in that it diverges substantially from the averaged results in Figure 5 of the original paper, which showed balanced utilization across racks. While the traffic is balanced across longer timescales, our results match more closely with their more granular findings of on-off periods and significant variance at medium timescales.

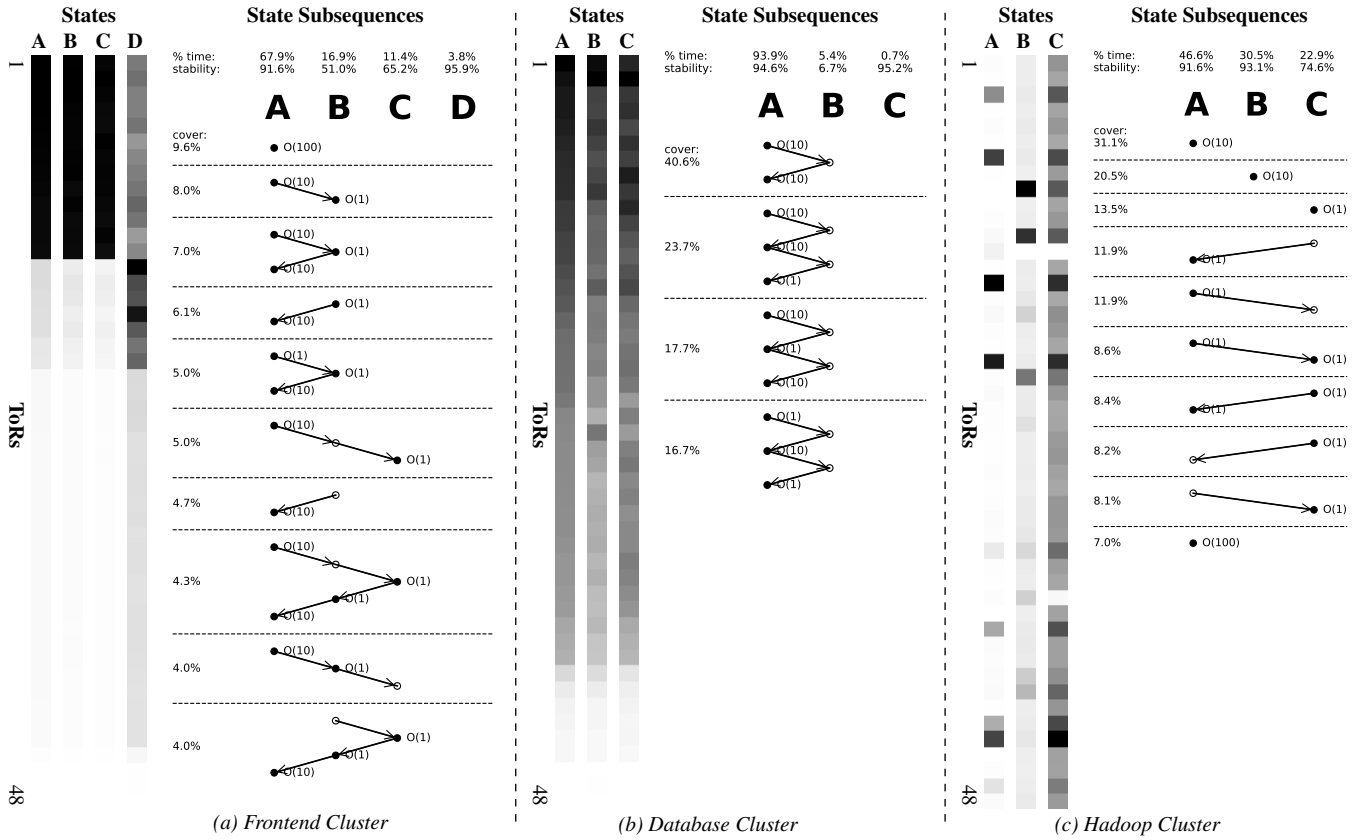
Signature	Accuracy	Precision	Recall
Straggler	0.943	0.867	0.720
Imbalance	0.936	1.000	0.868

Table 2: Classification performance of signatures in the memcached testbed.

7.3 Efficacy of the TPS Module

We showcase Traffic Pattern Scores by demonstrating how they can help answer an important question: *is my network performing poorly due to load imbalance or stragglers?* For this, we use the straggler and network imbalance signatures from Table 1 to diagnose issues in the memcache deployment from Section 5.3. We run the deployment in baseline, noisy neighbor, and ECMP misconfiguration scenarios. We then generate labeled network sample traces by manually identifying the precise time windows during which each undesired behavior occurred. Finally, we run `tpprof` on each of the traces and compare signature scores against ground truth scores calculated from sample labels.

Figure 14 plots the rolling average of ground truth and signature scores in each of the three scenarios. The signature scores are highly correlated with the ground truth. Table 2 lists the classification performance. Both signatures have high accuracy and precision, with slightly lower recall—a desirable tradeoff in an alerting system. We note that `tpprof`'s per-scenario precision and recall are 100%: no signature's score is high in the baseline scenario; only the straggler sig-



(a) Frontend Cluster *(b) Database Cluster* *(c) Hadoop Cluster*

Figure 13: `tpprof` profile of three 48-rack Facebook clusters. Figures include both (1) a collection of states (A–D) organized as a 1×48 heatmap, and (2) a list of the most common state subsequences. Letters map between the two representations.

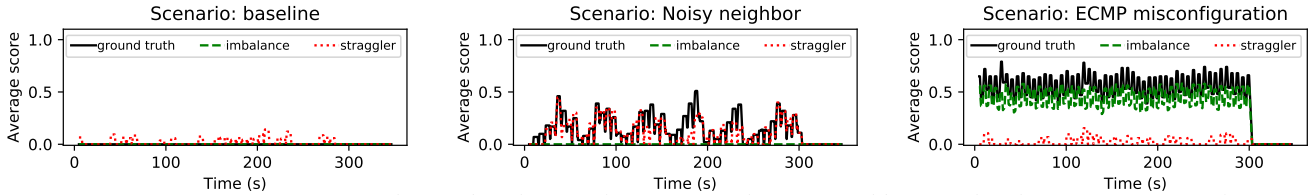


Figure 14: Signature scores for memcache in a baseline configuration, with noisy neighbors, and with an ECMP misconfiguration.

Filter count	0	128	256	512	1024
CPU Util (%)	0.44	0.65	0.84	1.18	1.77

Table 3: `tpprof`'s `iptables` CPU utilization.

nature's score is high in the noisy neighbor scenario; and only the imbalance signature's score is high in the ECMP misconfiguration scenario.

7.4 Overhead and Performance of `tpprof`

Finally, `tpprof` is designed for efficiency and minimal overhead. Only two components in the sampling framework can potentially impact traffic: sample collection and `iptables` tagging. Analytically, snapshots of all ports on a 128 port switch at a 50 ms interval generate only 0.1 Mb/s of measurement data. As Table 3 shows, the `iptables` rules used to construct application-specific profiles also have low overhead.

In addition to measuring overhead, we also benchmark the Hyperscan [34]-based TPS scoring engine, which operates online in parallel with the network. Specifically, we measure average CPU load while operating on the Facebook trace. Figure 15 shows single-core CPU load. It increases linearly with the number of signatures, but even in this large network with 100 signatures and a 50 ms sampling frequency, average load for real-time processing is only around 10%.

8 Related Work

Traffic pattern inference. We note that the concept of a network traffic pattern is not novel. Many prior works have both identified and used traffic patterns to great benefit [20, 29, 40, 54, 55, 67]. Unfortunately, these insights have typically been limited to situations where the pattern can be measured at a single link/device [40, 55, 67, 68] or have been a result of property-specific analyses, often with a large dose of manual effort [16, 20, 29, 54]. The goal of `tpprof` is instead

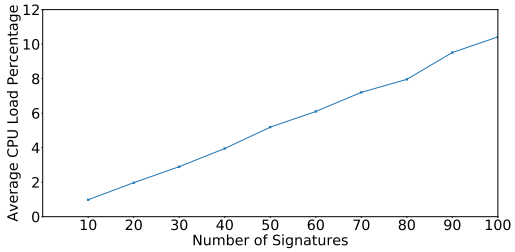


Figure 15: Signature vs CPU Load

the automatic extraction and ranking of common patterns from running networks.

Network monitoring and visualization tools. We also acknowledge the vast array of existing network monitoring and visualization tools, both commercial [1–5, 26, 37, 49, 52] and academic [30, 45, 47, 57, 61, 68]. We lack sufficient space to discuss them all, but one worth mentioning is Cisco's recent Tetration platform [52]. Among other features, Tetration can extract the control flow of a distributed application by clustering hosts based on the partners with which they communicate. Other work has attacked similar problems [36]. To the best of our knowledge, `tpprof` is the first tool that extracts common *network-wide traffic patterns*, rather than application-level communication patterns or packet/flow-level behavior. Broadly speaking, `tpprof` operates at a higher-level of abstraction than these existing systems.

Wee note, however, that `tpprof` is compatible with some infrastructure monitoring frameworks like Nagios [37] that collect monitoring data from across the network. By default, none of these provide the same abstraction as `tpprof`, but many allow custom measurement configurations and plugins, of which `tpprof` could be one.

Application performance profilers. Our work draws inspiration from a long history of work in application performance profiling [19, 22, 27, 46, 53, 60, 64]. Some of which are even able to profile distributed applications [32, 33, 42, 50]. While `tpprof` borrows its approach from the subset of these that profile stochastically, it does this for traffic patterns, which have their own unique set of challenges.

Anomaly detectors. Our alerting mechanisms are related to prior work in anomaly detection. Compared to unsupervised anomaly detection [18, 66], however, `tpprof` provides a much more accurate and fine-grained detection method. Compared to traditional profiling-based anomaly detection in which a user provides a 'correct' trace and the system determines whether the current system diverges [39, 59], `tpprof` can distinguish between different anomalies and does not require the user to obtain a correct trace. More generally, `tpprof`'s scoring engine presents a natural, declarative interface for the user tell the detector, via traffic pattern signatures, the approximate characteristics of relevant traffic patterns.

Clustering and compression. Finally, we note that our techniques for compressing network states borrow from or are related to the rich literature on clustering and compression [14, 23, 25, 35]. Our network state extraction techniques, in particular, leverage existing algorithms. The contribution of this work is instead the choice and tuning of these clustering algorithms to the domain of network traffic pattern analysis.

9 Discussion

Other metrics. While we focus on utilization in this paper, we note that `tpprof` easily extends to any metric collectible from the network. These include simple extensions like packet counts to more advanced metrics like buffer depth and high-water marks. As these metrics are generally correlated with utilization, we anticipate that `tpprof`'s techniques will extend intrinsically, but we leave an exploration of these extensions to future work.

Canned reactions. We also note that the ability of `tpprof`'s scoring engine to distinguish different traffic patterns presents an attractive substrate for building network-level reactions to different traffic patterns. This can also work in reverse: `tpprof` can identify common patterns for which operators should pre-compute reactions. We leave an investigation of this class of applications to future work as well.

10 Conclusion

We present `tpprof`, a network traffic pattern profiler. Just as tools like gprof made it easy for programmers to design, understand, and optimize their programs, `tpprof` does the same for profiling the utilization of large networks. `tpprof` leverages recent advancements in programmable networks and network-wide measurement to capture packet-accurate snapshots of utilization over time. On top of that, `tpprof` builds user-centric profiling, visualization, and automation tools. `tpprof` is agnostic to the application set running over the network and can profile networks in situ, making it an ideal fit for multi-tenant or transit networks. We profile several classic applications in order to demonstrate its utility.

Acknowledgments

We gratefully acknowledge Amin Vahdat for providing comments on drafts, our shepherd Andrew Moore, and the anonymous NSDI reviewers for all of their thoughtful reviews. This work was funded in part by Facebook, VMWare, NSF grant CNS-1845749, and DARPA contract HR0011-17-C0047.

References

[1] https://www.nutanix.com/products/epoch.

[2] https://www.pluribusnetworks.com/.

[3] https://www.logicmonitor.com/.

[4] https://endace.com.

[5] https://www.bigswitch.com/products/big-monitoring-fabric/.

[6] https://aws.amazon.com/blogs/security/tag/network-monitoring-tools/.

[7] http://docs.libmemcached.org/bin/memaslap.html.

[8] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[9] Dormando Anatoly Vorobey, Brad Fitzpatrick. Memcached, 2009.

[10] Apache Software Foundation. Giraph, 2012.

[11] Apache Software Foundation. Hadoop, terasort, 2012.

[12] Apache Software Foundation. Hadoop, yarn, 2012.

[13] Apache Software Foundation. Spark, 2016.

[14] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 476–489, New York, NY, USA, 2018. ACM.

[15] Richard E Bellman. *Adaptive control processes: a guided tour*. Princeton university press, 2015.

[16] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, January 2010.

[17] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In Catriel Beeri and Peter Buneman, editors, *Database Theory — ICDT'99*, pages 217–235, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[18] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: Identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, page 93–104, New York, NY, USA, 2000. Association for Computing Machinery.

[19] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[20] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 73–82, New York, NY, USA, 2009. ACM.

[21] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 31–36, New York, NY, USA, 2012. ACM.

[22] W.E. Cohen. Tuning programs with oprofile. *Wide Open Magazine*, 1:53–62, 01 2004.

[23] William HE Day and Herbert Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of classification*, 1(1):7–24, 1984.

[24] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 339–350, New York, NY, USA, 2010. ACM.

[25] Karl Pearson F.R.S. LIII. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[26] Gigamon. Security and networking solutions | gigamon, 2018.

[27] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[28] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 139–152, New York, NY, USA, 2015. ACM.

[29] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 38–49, New York, NY, USA, 2011. ACM.

[30] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation NSDI 14*, pages 71–85, Seattle, WA, 2014. USENIX Association.

[31] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[32] Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, and Timothy Roscoe. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 95–110, Renton, WA, 2018. USENIX Association.

[33] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. Distributed performance monitoring: methods, tools, and applications. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):585–598, June 1994.

[34] Intel. https://www.hyperscan.io/.

[35] Anil K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651 – 666, 2010. Award winning papers from the 19th International Conference on Pattern Recognition (ICPR).

[36] Yu Jin, Esam Sharafuddin, and Zhi-Li Zhang. Unveiling core network-wide communication patterns through application traffic activity graph decomposition. In *Proceedings of*

the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '09, page 49–60, New York, NY, USA, 2009. Association for Computing Machinery.

[37] David Josephsen. *Building a Monitoring Infrastructure with Nagios*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[39] Anukool Lakhina, Mark Crovella, and Christiphe Diot. Characterization of network-wide anomalies in traffic flows. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, page 201–206, New York, NY, USA, 2004. Association for Computing Machinery.

[40] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Trans. Netw.*, 2(1):1–15, February 1994.

[41] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[42] Inc. Lightstep. Lightstep [x]pm, 2019.

[43] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 399–412, Berkeley, CA, USA, 2013. USENIX Association.

[44] Vincent Liu, Danyang Zhuo, Simon Peter, Arvind Krishnamurthy, and Thomas Anderson. Subways: A case for redundant, inexpensive data center edge links. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 27:1–27:13, New York, NY, USA, 2015. ACM.

[45] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 101–114, New York, NY, USA, 2016. ACM.

[46] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 13–23, New York, NY, USA, 2005. ACM.

[47] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 129–143, New York, NY, USA, 2016. ACM.

[48] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[49] Barefoot Networks. Barefoot deep insight – product brief, 2018.

[50] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, Berkeley, CA, USA, 2015. USENIX Association.

[51] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[52] Remi Philippe. Next generation data center flow telemetry. Technical report, Cisco, 2016.

[53] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of win32/intel executables using etch. In *Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, NT'97, pages 1–1, Berkeley, CA, USA, 1997. USENIX Association.

[54] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 123–137, New York, NY, USA, 2015. ACM.

[55] Bong K. Ryu and Anwar Elwalid. The importance of long-range dependence of vbr video traffic in atm traffic engineering: Myths and realities. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '96, pages 3–14, New York, NY, USA, 1996. ACM.

[56] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a" kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st international conference on distributed computing systems workshops*, pages 166–171. IEEE, 2011.

[57] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[58] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. ACM.

[59] Ningombam Anandshree Singh, Khundrakpam Johnson Singh, and Tanmay De. Distributed denial of service attack detection

using naive bayes classifier through info gain feature selection. In *Proceedings of the International Conference on Informatics and Analytics*, ICIA-16, New York, NY, USA, 2016. Association for Computing Machinery.

[60] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 196–205, New York, NY, USA, 1994. ACM.

[61] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 561–575, New York, NY, USA, 2018. ACM.

[62] Nofel Yaseen, John Sonchack, and Vincent Liu. Speedlight bmv2, 2018.

[63] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 402–416, New York, NY, USA, 2018. ACM.

[64] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance analysis using the mips r10000 performance counters. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[65] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[66] Jiong Zhang and Mohammad Zulkernine. Anomaly based network intrusion detection with unsupervised outlier detection. *2006 IEEE International Conference on Communications*, 5:2388–2393, 2006.

[67] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, pages 78–85, New York, NY, USA, 2017. ACM.

[68] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick Duffield, and Carsten Lund. Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, pages 101–114, New York, NY, USA, 2004. ACM.

[69] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 479–491, New York, NY, USA, 2015. ACM.