# Programmable Host-Network Traffic Management

Peng Sun[†], Minlan Yu[‡], Michael J. Freedman[†], Jennifer Rexford[†], David Walker[†]
[†]Princeton University [‡]University of Southern California

## ABSTRACT

Data-center administrators perform traffic-management tasks (e.g., performance monitoring, server load balancing, and traffic engineering) to optimize network performance for diverse applications. Increasingly, traffic-management functionality is moving from the switches to the end hosts, which have more computational resources and better visibility into application behavior. However, traffic management is complicated by the heterogeneous interfaces for monitoring and controlling hosts and switches, and the scalability challenges of collecting and analyzing measurement data across the data center. We present a *scalable* and *programmable* platform for joint HOst-NEtwork (HONE) traffic management. HONE's programming environment gives a simple, integrated, and logically-centralized view of the data center for defining measurement, analysis, and control tasks across hosts and switches. Programmers can think globally and rely on HONE to distribute the program for local execution and scalable aggregation of data across hosts and switches. HONE successfully balances the inherent tension between ease-of-use and performance. We evaluate HONE by implementing several canonical traffic-management applications, measuring its efficiency with micro-benchmarks, and demonstrating its scalability with larger-scale experiments on Amazon EC2.

## 1. INTRODUCTION

Modern data centers run diverse applications that generate a large amount of network traffic. To optimize performance, data-center administrators perform many traffic-management tasks, such as performance monitoring, server load balancing, access control, flow scheduling, rate limiting, and traffic engineering. Increasingly, data-center traffic management capitalizes on the opportunity to move functionality from the switches to the end hosts [10, 14, 15, 24, 27, 28, 32, 34, 37]. Compared to hardware switches, hosts have better visibility into application behavior, greater computational resources, and more flexibility to adopt new functionality. By harnessing both the hosts and the switches, data-center administrators can improve application performance and make more efficient use of resources.

To build an effective joint host-network traffic-management system, we need to achieve three main goals:

**Uniform interface:** Data-center administrators use a variety of interfaces on hosts to collect socket logs, kernel statistics, and CPU/memory utilization (e.g., Windows ETW [1], Web10G [31], and vCenter [30]), and perform rate limiting, access control, and routing (e.g., Linux `tc` [16], `iptables` [19], and Open vSwitch [21]). Similarly, switches offer various measurement and control interfaces (e.g., NetFlow/sFlow, SNMP, OpenFlow, and command-line interfaces). Data-center administrators rely on numerous scripts and configuration files to automate their management tasks. A good management system would shield the administrators from these details by offering an *uniform* interface for measurement, analysis, and control across hosts and switches.

**Programmability:** Rather than settle on a single traffic-management solution in advance, a good system would be *programmable*, so administrators can support multiple management tasks at a time, and adopt new solutions as the application mix and network design evolve. The most general programming framework would allow administrators to deploy arbitrary code on the hosts, but this by itself does little to lower the barrier to creating new management applications. Instead, the system should raise the level of abstraction by offering a simple, logically-centralized view of the data center, and automatically handling the details of collecting and analyzing data, and enforcing control actions, across a distributed collection of hosts and switches.

**Efficiency and scalability:** Data centers have a large number of switches and hosts, each with many connections. Collecting and analyzing a diverse set of measurement from these devices in real time introduces a major scalability challenge. Parallelizing the collection and analysis of the data is crucial. Although automatically parallelizing an arbitrary program is notoriously difficult, a carefully designed programming language can expose opportunities to distribute the work over components in the system. In particular, the hosts should collect measurement data, and locally filter and aggregate the data, to reduce the load on the controller.

In this paper, we present a scalable, programmable

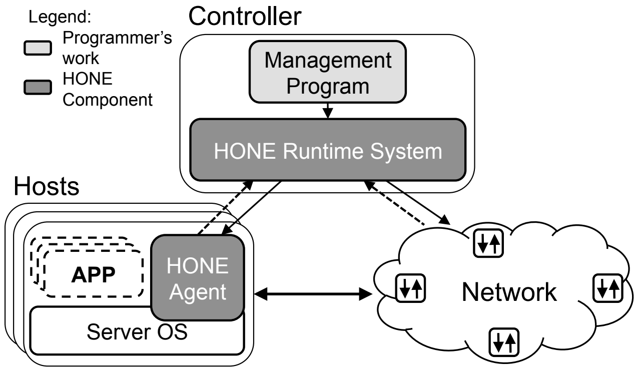


Figure 1: Overview of HONE system

platform for joint HOst-NEtwork (HONE) traffic management. As shown in Figure 1, a management program running on the controller defines what measurement data to collect, what analysis to perform, and what control actions to take. HONE's declarative programming environment enables programmers to think globally without worrying about how the hosts and switches act locally to execute the program. The controller divides the program logic, and scalably filters and aggregates the results. On each host, a HONE agent performs fine-grained measurement across several layers, and performs local analysis, at the behest of the controller.

To balance the inherent tension between ease-of-use and performance, we make several contributions:

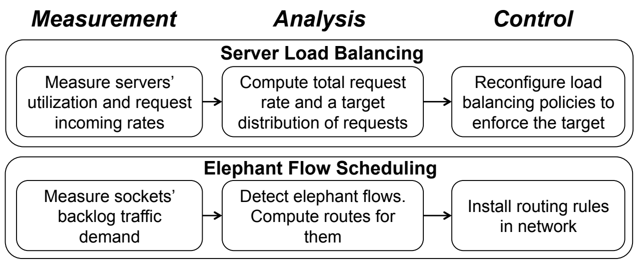
**Uniform host-network data model:** HONE abstracts the data center as a database, using global tables of statistics to uniformly represent data from a diverse array of sources. The controller handles the low-level details of collecting the necessary measurement data from hosts and switches using a variety of interfaces.

**Lazy materialization of measurement data:** The uniform data model, with diverse statistics available at arbitrary time granularity, is too expensive to support directly. Instead, the controller analyzes the queries to have the hosts collect only the necessary statistics for the appropriate connections and time periods.

**Uniform data-processing interface:** Programs analyze measurement data using data-parallel streaming operators, allowing programmers to think globally and sequentially. The controller handles the low-level details of distributing the computation, communicating with hosts and switches, and combining the results.

**Host-based filtering and aggregation:** The controller automatically partitions the analysis to use end-host resources to filter and aggregate the data locally. The controller groups hosts into a hierarchy to apply user-defined (commutative and associative) functions to combine analysis results across the data center.

**Case studies of management applications:** To demonstrate the power of our programming environment, we build a collection of canonical management applications, such as flow scheduling [3, 8], distributed


Measurement
Analysis
Control
Server Load Balancing
Measure servers' utilization and request incoming rates
Compute total request rate and a target distribution of requests
Reconfigure load balancing policies to enforce the target
Elephant Flow Scheduling
Measure sockets' backlog traffic demand
Detect elephant flows. Compute routes for them
Install routing rules in network
Figure 2: Three stages of traffic management

rate limiting [25, 26], network diagnosis [34], etc. These examples demonstrate the expressiveness of our language, as well as the scalability of our data collection and analysis techniques.

**Prototype implementation and evaluation:** Our prototype host agent collects socket logs and TCP statistics [17, 31], and performs traffic shaping using Linux `tc`. Our controller also monitors and configures switches using OpenFlow [18]. Experiments show that HONE can measure and calculate an application's throughput, and aggregate the results across 128 EC2 instances within a 90th-percentile latency of 58ms.

HONE's measurement and analysis framework has some similarities with streaming databases [4, 7] and MapReduce [9]. However, these general-purpose systems do not meet the needs of joint host-network traffic management in data centers. HONE enables programmers to collect diverse measurement data *on demand*, whereas streaming databases usually operate on streams that were defined *a priori* (e.g., sampled packets). Similarly, unlike HONE, MapReduce is not designed for processing data inherently tied to a specific host (e.g., socket logs and kernel statistics), nor for real-time analysis.

## 2. HONE PROGRAMMING FRAMEWORK

HONE's programming framework is designed to enable a wide range of traffic-management tasks. Traffic management is usually oriented around a three-stage "control loop" of measurement, data analysis, and control. Figure 2 presents two representative applications that serve as running examples throughout the paper:

**Server load balancing:** The first application distributes incoming requests across multiple server replicas. After measuring the request rate and the server load (e.g., CPU, memory, and bandwidth usages) at each host, the application estimates the total request rate, computes a new target division of requests over the hosts, and configures switches accordingly.

**Elephant flow scheduling:** The second application is inspired by how Hedera [3] and Mahout [8] schedule large flows. After measuring the backlog in the socket buffer for each TCP connection, the application identifies the elephant flows and directs them over paths that minimize network congestion.

### 2.1 Measurement: Query on Global Tables

| Table Name | Row for each | Columns |
|---|---|---|
| Connections | Connection | App, TCP/UDP five-tuple, TCP-stack statistics. |
| Applications | Process | Host ID, PID, app's name, CPU/memory usages. |
| Machines | Host | Host ID, total CPU usage, total memory usage, IP. |
| Links | Link | IDs/ports of two ends, capacity. |
| SwitchStats | Switch interface | Switch ID, port, timestamp, per-port counters. |

Table 1: Global tables supported in HONE prototype

HONE's data model unifies the representation of statistics across a range of formats, locations, types of devices, and modes of access. The HONE controller offers a simple abstraction of a central set of database tables. Programmers can launch sophisticated queries, and rely on HONE to distribute the monitoring to the devices, materialize the necessary tables, transform the data to fit the schema, perform local computations and data reduction, and aggregate the data. The data model reduces a complex and error-prone distributed programming task to a set of simple, tabular queries that can usually be crafted in just tens of lines of code.

The HONE data model is organized around the protocol layers and the available data sources. Table 1 shows the tables that our current prototype supports. On the hosts, HONE collects socket logs and TCP connection statistics, to capture the relationship between applications and the network stack while remaining application agnostic. On the switches, HONE collects the topology, the routing configurations, and per-port counters using OpenFlow. However, we can easily extend our prototype to support more interfaces (e.g., NetFlow) by adding new tables, along with implementations for collecting the data.

HONE offers programmers a familiar, SQL-like query language for collecting the data, as summarized in Table 2. The query language gives programmers a way to state declaratively *what* data to measure, rather than *how*. More sophisticated analysis, transformation, filtering, and aggregation of the data take place in the analysis phase. To illustrate how to create a HONE program, consider the three example queries needed for elephant-flow scheduling:

**Backlog in socket buffer:** This query generates the data for computing the backlog in the socket buffers:

```
def ElephantQuery():
  return (
    Select([SrcIp, DstIp, SrcPort, DstPort,
            BytesWritten, BytesSent]) *
    From(Connections) *
    Every(Seconds 1) )
```

The query produces a stream of tables, with one table every second.[1] In each table, each row corresponds to a single connection and contains the endpoint IP addresses and port numbers, as well as the amount of data written into the socket buffer and sent into the network. Later, the analysis phase can use the per-connection BytesWritten and BytesSent to compute the backlog in the socket buffer to detect elephant flows.

| | | |
|---|---|---|
| Query | := | *Select*(Stats) * *From*(Table) * *Where*(Criteria) * *Groupby*(Stat) * *Every*(Interval) |
| Table | := | Connections \| Applications \| Links \| SwitchStats \| Machines |
| Stats | := | *Columns* of Table |
| Interval | := | *Integer* in Seconds or Milliseconds |
| Criteria | := | Stat Sign *value* |
| Sign | := | $> \mid < \mid \geq \mid \leq \mid = \mid \neq$ |

Table 2: Measurement query language syntax

**Connection-level traffic by host pair:** This query collects the data for computing the traffic matrix:

```
def TrafficMatrixQuery():
  return(
    Select([SrcIp, DstIp, BytesSent,
            Timestamp]) *
    From(Connections) *
    Groupby([SrcIp,DstIp]) *
    Every(Seconds 1) )
```

The query uses the Groupby operator to convert each table (at each second) into a *list* of tables, each containing information about all connections for a single pair of end-points. Later, the analysis phase can sum the BytesSent across all connections in each table in the list, and compute the difference from one time period to the next to produce the traffic matrix.

**Active links and their capacities:** This query generates a stream of tables with all unidirectional links in the network:

```
def LinkQuery():
  return(
    Select([BeginDevice, EndDevice,
            Capacity]) *
    From(Links) *
    Every(Seconds 1) )
```

Together, these queries provide the information needed for the elephant-flow application. They also illustrate the variety of different statistics that HONE can collect from both hosts and switches—all within a simple, unified programming framework. Under the hood, the HONE runtime system keeps track of which data should come from each host and each switch (to minimize the data-collection overhead) and parallelize the analysis.

[1]The star operator (*) glues together the various query components. Each query term generates a bit of abstract syntax that our runtime system interprets.

## 2.2 Analysis: Data-Parallel Operators

HONE enables programmers to analyze data across multiple hosts, without worrying about the low-level details of communicating with the hosts or tracking their failures. HONE's functional data-parallel operators allow programmers to say *what* analysis to perform, rather than *how*. Programmers can associate their own functions with the operators to apply these functions across sets of hosts, as if the streams of tabular measurement data were all available at the controller. Yet, HONE gives the programmers a way to express whether their functions can be (safely) applied in parallel across data from different hosts, to enable the runtime system to reduce the bandwidth and processing load on the controller by executing these functions at the hosts. HONE's data-parallel operators include the following:

- `MapSet(`$f$`)`: Apply function $f$ to every element of a stream in the set of streams, producing a new set of streams.
- `FilterSet(`$f$`)`: Create a new set of streams that omits stream elements $e$ for which $f(e)$ is false.
- `ReduceSet(`$f$`,`$i$`)`: "Fold" function $f$ across each element for each stream in the set, using $i$ as an initializer. In other words, generate a new set of streams where $f(\ldots f(f(i, e_1), e_2)\ldots, e_n)$ is the $n^{th}$ element of each stream when $e_1$, $e_2$, ..., $e_n$ were the first $n$ elements of the original stream.
- `MergeHosts()`: Merge a set of streams on the hosts into one single global stream. (Currently in HONE, the collection of switches already generate a single global stream of measurement data, given that our prototype integrates with an SDN controller to access data from switches.)

`MapSet`, `FilterSet`, and `ReduceSet` operate in parallel on each host, and `MergeHosts` merges the results of multiple analysis streams into a single stream on the controller. HONE also enables analysis on a single global stream with corresponding operators, such as `MapStream`, `FilterStream`, and `ReduceStream`. To combine queries and analysis into a single program, the programmer simply associates his functions with the operators, and "pipes" the result from one query or operation to the next (using the >> operator).

Consider again the elephant-flow scheduling application, which has three main parts to the analysis:

**Identifying elephant flows:** Following the approach suggested by Curtis *et al.* [8], the function `IsElephant` defines elephant flows as the connections with a socket backlog (i.e., the difference between bytes `bw` written by the application and the bytes `bs` acknowledged by the recipient) in excess of 100KB:

```
def IsElephant(row):
  [sip,dip,sp,dp,bw,bs] = row
  return (bw-bs > 100)
```

```
def DetectElephant(table):
  return (FilterList(IsElephant, table))

EStream = ElephantQuery()         >>
          MapSet(DetectElephant) >>
          MergeHosts()
```

`DetectElephant` uses `FilterList` (the same as `filter` in Python) to apply `IsElephant` to select only the rows of the connection table that satisfy this condition. Finally, `DetectElephant` is applied to the outputs of `ElephantQuery`, and the results are merged across all hosts to produce a single stream `EStream` of elephant flows at the controller.

**Computing the traffic matrix:** The next analysis task computes the traffic matrix, starting from aggregating the per-connection traffic volumes by source-destination pair, and then computing the difference across consecutive time intervals:

```
TMStream = TrafficMatrixQuery()   >>
           MapSet(MapList(SumBytesSent) >>
           ReduceSet(CalcThroughput, {}) >>
           MergeHosts()           >>
           MapStream(AggTM)
```

The query produces a stream of lists of tables, where each table contains the per-connection traffic volumes for a single source-destination pair at a point in time. `MapList` (i.e., the built-in `map` in Python) allows us to apply a custom function `SumBytesSent` that aggregates the traffic volumes across connections in the same table, and `MapSet` applies this function over time. The result is a stream of tables, which each contains the cumulative traffic volumes for every source-destination pair at a point in time. Next, the `ReduceSet` applies a custom function `CalcThroughput` to compute the differences in the total bytes sent from one time to the next. The last two lines of the analysis merge the streams from different hosts and apply a custom function `AggTM` to create a global traffic matrix for each time period at the controller.

**Constructing the topology:** A last part of our analysis builds a network topology from the link tables produced by `LinkQuery`, which is abstracted as a single data stream collected from the network:

```
TopoStream = LinkQuery() >>
             MapStream(BuildTopo)
```

The auxiliary `BuildTopo` function (not shown) converts a single table of links into a graph data structure useful for computing paths between two hosts. The `MapStream` operator applies `BuildTopo` to the stream of link tables to generate a stream of graph data structures.

## 2.3 Control: Uniform and Dynamic Policy

In controlling hosts and switches, the data-center administrators have to use various interfaces. For example, administrators use `tc`, `iptables`, or Open vSwitch

| | |
|---|---|
| Policy | := [Rule]+ |
| Rule | := *if* Criteria *then* Action |
| Criteria | := Predicate [(and \| or) Predicate]* |
| Predicate | := Field = *value* |
| Field | := AppName \| SrcHost \| DstHost \| Headers |
| Headers | := SrcIP \| DstIP \| SrcPort \| DstPort \| ··· |
| Action | := rate-limit *value* \| forward-on-path *path* |

Table 3: Control policy supported in HONE prototype

on hosts to manage traffic, and they use SNMP or OpenFlow to manage the switches. For the purpose of managing traffic, these different control interfaces can be unified because they share the same pattern of generating control policies: for a group of connections satisfying criteria, define what actions to take. Therefore, HONE offers administrators a uniform way of specifying control policies as *criteria + action* clauses, and HONE takes care of choosing the right control implementations, e.g., we implement *rate-limit* using tc and iptables in the host agent.

The criteria can be network identifiers (e.g., IP addresses, port numbers, etc.). But this would force the programmer to map his higher-level policies into lower-level identifiers, and identify changes in which connections satisfy the higher-level policies. Instead, we allow programmers to identify connections of interest based on higher-level attributes, and HONE automatically tracks which traffic satisfies these attributes as connections come and go. Our predicates are more general than network-based rule-matching mechanisms in the sense that we can match connections by applications with the help of hosts. Table 3 shows the syntax of control policies, each of which our current prototype supports.

Continuing the elephant-flow application, we define a function Schedule that takes inputs of the detected elephant flows, the network topology, and the current traffic matrix. It assigns a routing path for each elephant flow with a greedy *Global First Fit* [3] strategy, and creates a HONE policy for forwarding the flow along the picked path. Other non-elelphant flows are randomly assigned to an available path. The outputs of policies by Schedule will be piped into RegisterPolicy to register them with HONE.

```
def Schedule(elephant, topo, traffic):
  routes = FindRoutesForHostPair(topo)
  policies = []
  for four_tuples in elephant:
    path = GreedilyFindAvailablePath(
           four_tuples, routes, traffic)
    criteria = four_tuples
    action = forward-on-path path
    policies.append([criteria, action])
  return policies
```

## 2.4 All Three Stages Together

Combining the measurement, analysis, and control phases, the complete program merges the data streams, feeds the data to the Schedule function, and registers the output of policies. With this concrete example of an elephant-flow detection and scheduling application, we have demonstrated the simple and straightforward way of designing traffic-management tasks in HONE programming framework.

```
def ElephantFlowDetectionScheduling():
  MergeStreams (
    [EStream, TopoStream, TMStream]) >>
  MapStream(Schedule)                >>
  RegisterPolicy()
```

## 3. EFFICIENT & SCALABLE EXECUTION

Monitoring and controlling many connections for many applications on many hosts could easily overwhelm a centralized controller. HONE overcomes this scalability challenge in four main ways. First, a distributed directory service dynamically tracks the mapping of management tasks to hosts, applications, and connections. Second, the HONE agents lazily materialize virtual tables based on the current queries. Third, the controller automatically partitions each management task into global and local portions, and distributes the local part over the host agents. Fourth, the hosts automatically form a tree to aggregate measurement data based on user-defined aggregation functions to limit the bandwidth and computational overhead on the controller.

### 3.1 Distributed Directory Service

HONE determines which hosts should run each management task, based on which applications and connections match the queries and control policies. HONE has a directory service that tracks changes in the active hosts, applications, and connections. To ensure scalability, the directory has a two-tiered structure where the first tier (tracking the relatively stable set of active hosts and applications) runs on the controller, and the second tier (tracking the large and dynamic collection of connections) runs locally on each host. This allows the controller to decide which hosts to inform about a query or control policy, while relying on each local agent to determine which connections to monitor or control.

**Tracking hosts and applications:** Rather than build the first tier of the directory service as a special-purpose component, we leverage the HONE programming framework to run a standing query:

```
def DirectoryService():
  (Select([HostID, App]) *
   From(Applications) *
   Every(Seconds 1) )                     >>
  ReduceSet(GetChangeOfAppAndHealth,[]) >>
  MergeHosts()                            >>
  MapStream(NotifyRuntime)
```

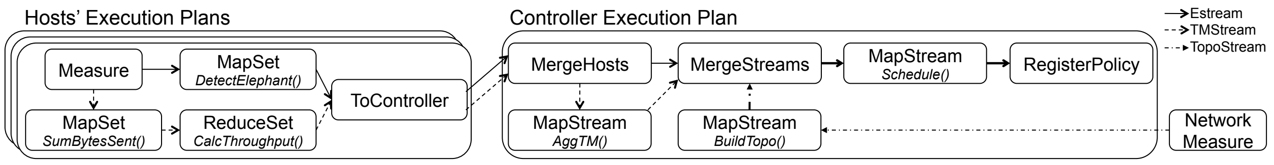


Figure 3: Partitioned execution plan of elephant-flow scheduling program

which returns the set of active hosts and their applications. GetChangeOfAppAndHealth identifies changes in the set of applications running on each host, and the results are aggregated at the controller. The controller uses its connectivity to each host agent as the host's health state, and the host agent uses ps to find active applications.

**Tracking connections:** To track the active connections, each host runs a Linux kernel module we build that intercepts the socket system calls (i.e., connect, accept, send, receive, and close). Using the kernel module, the HONE agent associates each application with the TCP/UDP connections it opens in an event-driven fashion. This avoids the inevitable delay of poll-based alternatives using Linux lsof and /proc.

## 3.2 Lazily Materialized Tables

HONE gives programmers the abstraction of access to diverse statistics at any time granularity. To minimize measurement overhead, HONE lazily materializes the statistics tables by measuring only certain statistics, for certain connections, at certain times, as needed to satisfy the queries. Returning to the elephant-flow application, the controller analyzes the ElephantQuery and decides to run the query on the hosts. Since the query does not constrain the set of hosts and applications, the controller instructs *all* local agents to run the query.

Each HONE agent periodically measures the values of SrcIP, DstIP, SrcPort, DstPort, and BytesSent from the network stack (via Web10G [31]), and collects the BytesWritten from the kernel module discussed earlier in §3.1. Similarly, HONE queries the switches for the LinkQuery data; in our prototype, we interact with network devices using the OpenFlow protocol. HONE does not collect or record any unnecessary data. Lazy materialization supports a simple and uniform data model while keeping measurement overhead low.

## 3.3 Host-Controller Partitioning

In addition to selectively collecting traffic statistics, the hosts can significantly reduce the resulting data volume by filtering or aggregating the data. For example, the hosts could identify connections with a small congestion window, sum throughputs over all connections, or find the top $k$ flows by traffic volume.

However, parallelizing an arbitrary controller program would be difficult. Instead, HONE provides a MergeHosts operator that explicitly divides a task into its local and global parts. Analysis functions before MergeHosts run locally on each host, whereas functions after MergeHosts run on the controller. HONE hides the details of distributing the computation, communicating with end devices, and merging the results. Having an explicit MergeHosts operator obviates the need for complex code analysis for automatic parallelization.

HONE coordinates the parallel execution of tasks across a large group of hosts.[2] We first carry out industry-standard clock synchronization with NTP [20] on all hosts and the controller. Then the HONE runtime stamps each management task with its creation time $t_c$. The host agent dynamically adjusts when to start executing the task to time $t_c + nT + \epsilon$, where $n$ is an integer, $\epsilon$ is set to 10ms, and $T$ is the period of task (as specified by the Every statement). Furthermore, the host agent labels the local execution results with a logical sequence number (i.e., $n$), in order to tolerate the clock drift among hosts. The controller buffers and merges the data bearing the same sequence number into a single collection, releasing data to the global portion of task when either receiving from all expected hosts or timing out after $T$.

Using our elephant-flow-scheduling application, Figure 3 shows the partitioned execution plan of the management program. Recall that we merge EStream, TMStream, and TopoStream to construct the program. The measurement queries are interpreted as parallel *Measure* operations on the host agents, and the query of switch statistics from the network module. HONE executes the EStream and TMStream tasks on each host in parallel (to detect elephant flows and calculate throughputs, respectively), and streams these local results to the controller (i.e., *ToController*). The merged local results of TMStream pass through a throughput aggregation function (*AggTM*), and finally merge together with the flow-detection data and the topology data from TopoStream to feed the *Schedule* function.

## 3.4 Hierarchical Data Aggregation

Rather than transmit (filtered and aggregated) data directly to the controller, the hosts construct a hierarchy to combine the results using user-specified functions. HONE automatically constructs a $k$-ary tree

[2]The HONE controller ships the source code of the local portion of management tasks to the host agent. Since HONE programs are written in Python, the agent can execute them with its local Python interpreter, and thus avoids the difficulties of making the programs compatible with diverse environments on the hosts.

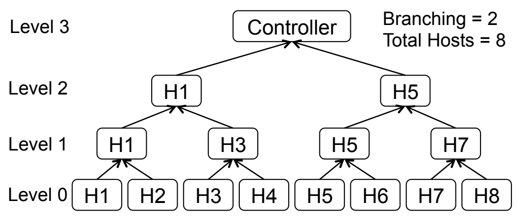


Figure 4: Aggregation tree: 8 hosts in branching of 2

rooted at the controller[3] and applies a TreeMerge operator at each level. All hosts running the task are leaves of the tree. For each group of $b$ hosts, HONE chooses one to act as their parent in the tree. These parents are grouped again to recursively build the tree towards the controller. User-defined functions associated with TreeMerge are applied to all non-leaf nodes of the tree to aggregate data from their children. HONE is unique among research efforts on tree-based aggregation [29, 33], since prior works focus on aggregating data with *a priori* knowledge of the data structure, and don't allow users to specify their own aggregation functions.

Many aggregation functions used in traffic management are both commutative and associative; such functions can be applied hierarchically without compromising correctness. For example, determining the top $k$ values for heavy-hitter analysis is amenable to either direct processing across all data or to breaking the data into subsets for intermediate analysis and combining the results downstream. Calculating the total throughput of connections across all hosts can also be calculated in such a distributed manner, as the arithmetic sum is also a commutative and associative function.

Making the user-defined aggregation functions be associative and commutative ensures that HONE can apply them correctly in a hierarchical manner. Using TreeMerge, HONE assumes that the associated functions have the required properties, avoiding the semantics analysis. TreeMerge is similar to MergeHosts in the sense that they both combine local data streams from multiple hosts into one data stream on the controller, and intermediate hosts similarly buffer data until they receive data from all their children or a timeout occurs. But with TreeMerge, HONE also applies a user-defined aggregation function, while MergeHosts simply merges all hosts' data at the controller without reduction.

The algorithm of constructing the aggregation tree is an interesting extensible part of HONE. We can group hosts based on their network locality, or we can dynamically monitor the resource usages on hosts to pick the one with most available resource to act as the intermediate aggregator. In our prototype, we leave those interesting algorithms to future works, but offer a basic one of incrementally building the tree by when hosts join the HONE system. Subject to the branching factor $b$, the newly joined leaf greedily finds a node in one level up with less than $b$ children, and links with the node if found. If not found, the leaf promotes itself to one level up, and repeats the search. When the new node reaches the highest level and still cannot find a place, the controller node moves up one level, which increases the height of the aggregation tree. Figure 4 illustrates an aggregation tree under the basic algorithm when 8 hosts have joined and $b$ is 2.

[3]The runtime uses information from the directory service to discover and organize hosts.

## 4. PERFORMANCE EVALUATION

In this section, we present micro-benchmarks on our HONE prototype to evaluate measurement overhead, the execution latency of management programs, and the scalability; §5 will demonstrate the expressiveness and ease-of-use of HONE using several canonical traffic-management applications.

We implement the HONE prototype in combination of Python and C. The HONE controller provides the programming framework and runtime system, which partitions the management programs, instructs the host agents for local execution, forms the aggregation hierarchy, and merges the data from hosts for the global portion of program execution. The host agent schedules the installed management tasks to run periodically, executes the local part of the program, and streams the serialized data to the controller or intermediate aggregators. We implement the network part of the prototype as a custom module in Floodlight [11] to query switch statistics and install routing rules.

Our evaluation of the prototype focuses on the following questions about our design decisions in §2 and §3.

1. How efficient is the host-based measurement in HONE?
2. How efficiently does HONE execute entire management tasks?
3. How much overhead does lazy materialization save?
4. How effectively does the controller merge data from multiple hosts using hierarchical aggregation?

We run the HONE prototype and carry out the experiments on Amazon EC2. All instances have 30GB memory and 8 virtual cores of 3.25 Compute Units each.[4]

### 4.1 Performance of Host-Based Measurement

The HONE host agent collects TCP connection statistics using the Web10G [31] kernel module. We evaluate the measurement overhead in terms of time, CPU, and memory usage as we vary the number of connections running on the host. To isolate the measurement overhead, we run a simple management task that queries a few randomly-chosen statistics of all connections running on the host every one second (we choose the four

[4]One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or Xeon processor.

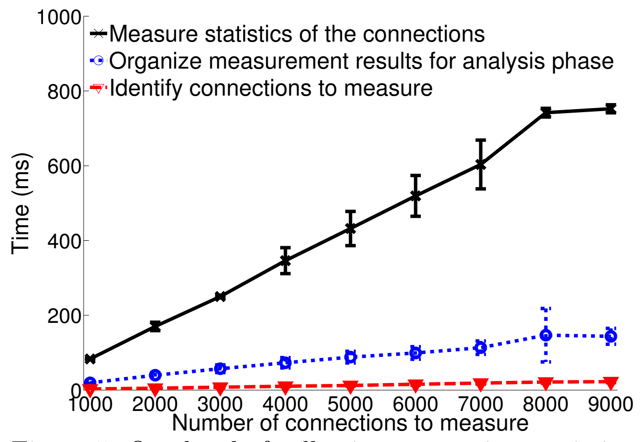


Figure 5: Overhead of collecting connection statistics

tuples, bytes of sent data, and the congestion window size). Our experiment consists of three EC2 instances—one for the controller, and two running the HONE agent.

To collect the statistics, the host agent must first identify what connections to measure. Then the agent queries the kernel via Web10G to retrieve the statistics. Finally, the agent organizes the statistics in the schema specified by the query and feeds the result to the management program. In Figure 5, we break down the latency in each portion. For each fixed number of connections, we run the management task for five minutes (i.e., about 300 iterations), and plot the average and standard deviation of time spent in each portion.

Figure 5 shows that the agent performs well, measuring 5k connections in an average of 532.6ms. The Web10G measurement takes the biggest portion–432.1ms, and the latency is linear in the number of active connections. The time spent in identifying connections to measure is relatively flat, since the agent tracks the relevant connections in an event-driven fashion via the kernel module of intercepting socket calls. The time spent in organizing the statistics rises slowly as the agent must go through more connections to format the results into the query's schema. The results set lower limit for the periods of management tasks that need measurement of different numbers of connections. The CPU and memory usages of the agent remain stable throughout the experiments, requiring an average of 4.55% CPU of one core and 1.08% memory of the EC2 instance.

## 4.2 Performance of Management Tasks

Next, we evaluate the end-to-end performance of several management tasks. To be more specific, we evaluate the latency of finishing one *round* of a task: from the agent scheduling a task to run, measuring the corresponding statistics, finishing the local analysis, sending the results to the controller, the controller receiving the data, till the controller finishing the remaining parts of the management program. We run three different kinds of management tasks which have a mix of leverages of

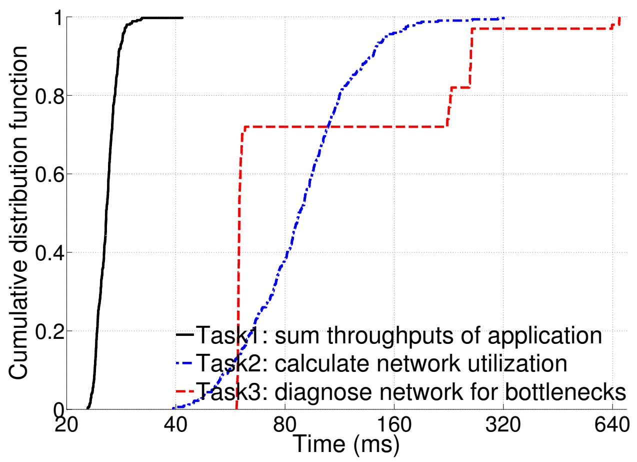


Figure 6: Latency of one round of execution

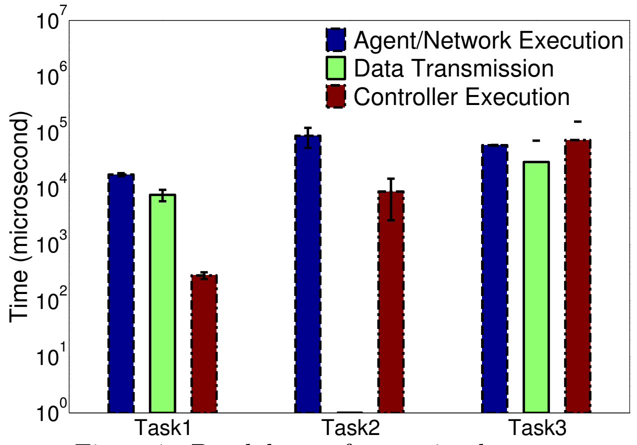


Figure 7: Breakdown of execution latency

hosts, switches, and the controller in HONE, in order to show the flexibility of HONE adapting to different traffic-management tasks. All experiments in this subsection run on a 8-host-10-switch fat-tree topology [2]. The switches are emulated by running Open vSwitch on an EC2 instance.

- ***Task1*** calculates the throughputs of all iperf connections on each host, sums them up, and aggregates the global iperf throughput at the controller. This task performs most of the analysis at the host agents, leaving relatively little work for the controller. Each host launches 100 iperf connections to another randomly chosen host.
- ***Task2*** queries the topology and switch statistics from the network, and uses the per-port counters on the switches to calculate the current link utilization. This task uses the network module in HONE a lot to measure data, and runs computation work on the controller. *Task2* is performed under the same setting of running iperf as *Task1*.
- ***Task3*** collects measurement data from the hosts to detect connections with a small congestion window (i.e., which perform badly). It also queries the net-

|  | **CPU Agent** | **Mem Agent** | **CPU Controller** | **Mem Controller** |
|---|---|---|---|---|
| Task1 | 3.71% | 0.94% | 0.67% | 0.10% |
| Task2 | N/A | N/A | 0.76% | 1.13% |
| Task3 | 7.84% | 1.64% | 1.03% | 0.11% |

Table 4: Avg. CPU and memory usages of execution

work to determine the forwarding path for each host pair. The task then diagnoses the shared links among those problematic flows as possible causes of the bad network performance. *Task3* is a joint host-network job, which runs its computation across hosts, network, and the controller. *Task3* is still under the same setting, but we manually add rules on two links to drop 50% of packets for all flows traversing the links, emulating a lossy network.

Figure 6 illustrates the cumulative distribution function (CDF) of the latency for finishing one round of execution, as we run 300 iterations for each task. We further break down the latency into three parts: the execution time on the agent or the network, the data-transmission time from the host agent or network module to the controller, and the execution time on the controller. In Figure 7, we plot the average latency and standard deviation for each part of the three tasks. *Task1* finishes one round with a 90th-percentile latency of 27.8ms, in which the agent takes an average of 17.8ms for measurement and throughput calculation, the data transmission from 8 hosts to the controller takes another 7.7ms, and the controller takes the rest. Having a different pattern with *Task1*, *Task2*'s 140.0ms 90th-percentile latency is consisted of 87.5ms of querying the switches via Floodlight and 8.9ms of computation on the controller (the transmission time is near zero since Floodlight is running on the controller machine). *Task3*'s latency increases as it combines the data from both hosts and the network, and its CDF also has two stairs due to different responsiveness of the host agents and the network module.

Table 4 summarizes the average CPU and memory usages on the host agent and the controller when running the task. The CPU percentage is for one core of 8 cores of our testbed machines. The results show that HONE's resource usages are bind to the running management tasks: *Tasks3* is the most complex one with flow detection/rate calculation on the hosts, and having the controller join host and network data.

## 4.3 Effects of Lazy Materialization

HONE lazily materializes the contents of the statistics tables. We evaluate how much overhead the feature can save for measurement efficiency in HONE.

We set up two applications ($A$ and $B$) with 1k active connections each on a host. We run multiple ma-

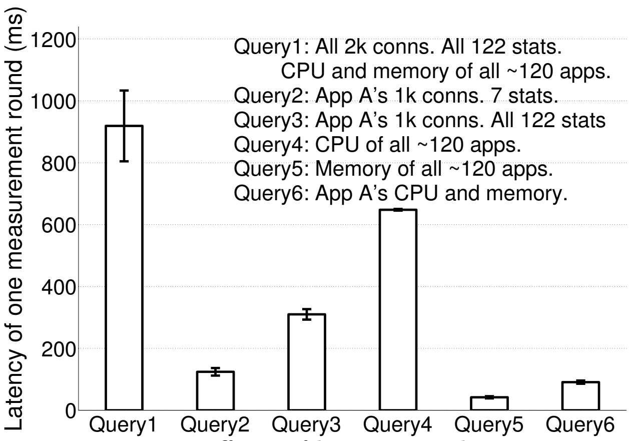


Figure 8: Effects of lazy materialization

nagement tasks with different queries over the statistics to evaluate the measurement overhead in terms of latency. Figure 8 illustrates the average and standard deviation of the latencies for different queries. The first program queries all 122 TCP-stack statistics available in Web10G of all 2k connections, and all applications' CPU and memory usages. The following ones query various statistics of `Connections` or `Applications` tables with details shown on Figure 8.

The lazy materialization of the tables lowers the measurement overhead by either measuring a subset of tables (*Query1* vs. others), rows (number of connections in *Query1* vs. *Query2* and *Query3*), and columns (number of statistics in *Query2* vs. *Query3*). The high overhead of *Query4* is due to the implementation of CPU measurement, which is, for each process, one of the ten worker threads on the agent keeps running for 50ms to get a valid CPU usage.

## 4.4 Evaluation of Scalability in HONE

We will evaluate the scalability of HONE from two perspectives. First, when HONE controller partitions the management program into local and global parts of execution, the controller will handle the details of merging the local results processed in the same time period from multiple hosts, before releasing the merged result to the global part of execution. Although the host clocks are synchronized via NTP as mentioned in §3.3, the clocks still drift slightly over time. It results in a buffering delay at the controller. Now we will evaluate how well the buffering works in terms of the time difference between when the controller receives the first piece of data and when the controller receives all the data bearing the same sequence number.

To focus on the merging performance, we use the *Task1* in §4.2. All hosts will directly send their local results to the controller without any hierarchical aggregation. Each run of the experiment lasts 7 minutes, containing about 400 iterations. We repeat the experiment, varying the number of hosts from 16 to 128.

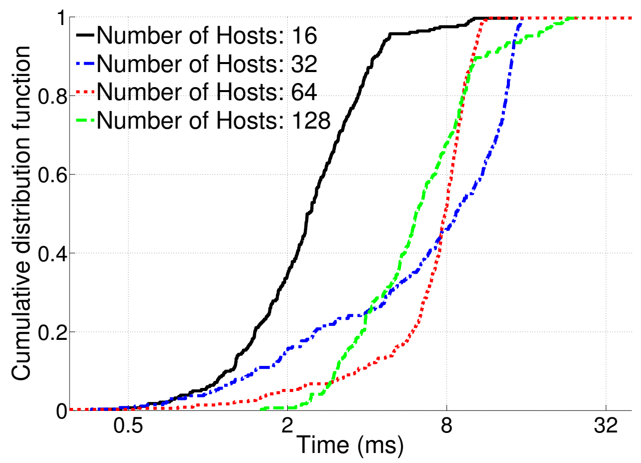


Figure 9: Controller's buffering delay of merging data from multiple host agents

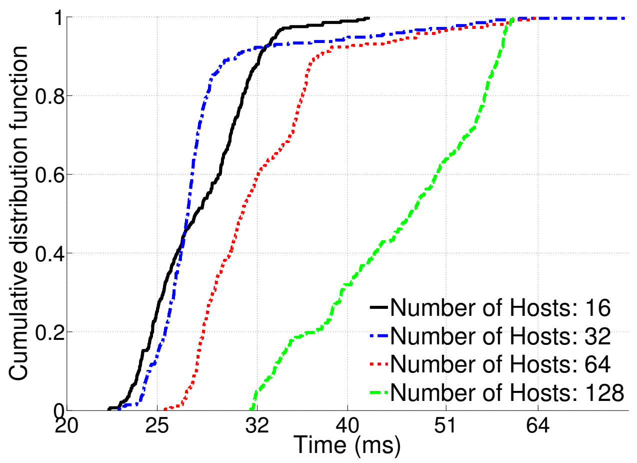


Figure 10: End-to-end latency of execution under hierarchical aggregation

Figure 9 shows the CDFs of the latencies for these experiments. The 90th-percentile of the controller's buffering delay is 4.3ms, 14.2ms, 9.9ms, and 10.7ms for 16, 32, 64, and 128 hosts respectively. The results show that the synchronization mechanism on host agents work well in coordinating their local execution of a management task, and the controller's buffering delay is not a problem in supporting traffic-management tasks whose execution periods are typically in seconds.

After evaluating how the controller merges distributed collection of data, we would evaluate another important feature of HONE for scalability, which is the hierarchical aggregation among the hosts. We continue using the same management task of summing the application's throughputs across hosts. But we change to use the TreeMerge operator to apply the aggregation function. In this way, the task will be executed by HONE through a $k$-ary tree consisted of the hosts.

In this experiment, we fix the branching factor $k$ of the hierarchy to 4. We repeat the experiment with 16, 32, 64, and 128 hosts, in which case the height of the aggregation tree is 2, 3, 3, and 4 respectively. Figure 10 shows the CDFs of the latencies of one round of execution, which captures the time difference from the earliest agent starting its local part to the controller finishing the global part. The 90th-percentile execution latency increases from 32.2ms, 30.5ms, 37.1ms, to 58.1ms. Table 5 shows the average CPU and memory usages on the controller and the host agent. The host agent's CPU and memory usages come from the agent that multiplexes as local-data generator and the intermediate aggregators in all levels of the $k$-ary tree. It shows the maximum overhead that the host agent incurs when running in a hierarchy.

From the results above, we can conclude that HONE's own operations pose little overhead to the execution of management tasks. The performance of management tasks running in HONE will be mainly bound by their own program complexities, and the amount of data they need to process or transmit.

| Num of Hosts | CPU Agent | Mem Agent | CPU Controller | Mem Controller |
|---|---|---|---|---|
| 16 | 4.19% | 0.96% | 1.09% | 0.05% |
| 32 | 4.93% | 0.96% | 1.27% | 0.05% |
| 64 | 5.26% | 0.97% | 1.31% | 0.06% |
| 128 | 4.80% | 0.97% | 2.36% | 0.07% |

Table 5: Avg. CPU and memory usages in hierarchical aggregation experiments

## 5. CASE STUDIES

We have shown the micro-benchmark evaluation of HONE to demonstrate its efficiency and scalability. Now we will illustrate the expressiveness and ease-of-use of HONE by building a diversity of traffic-management tasks in data centers. Table 6 lists all the management tasks that we have built, ranging from conventional management operations in data centers (e.g., calculating link utilizations) to recent proposals (e.g., network performance diagnosis [34]). Those conventional traffic-management tasks can actually serve as basic building blocks for more complex management tasks. The administrators can compose the code of those HONE programs to construct their own. HONE is an open-source project, and code for the management programs are also available at `http://hone.cs.princeton.edu/examples`.

In the following subsections, we pick two management tasks as case studies to illustrate more details, and evaluate the HONE-based solutions.

### 5.1 Elephant Flow Scheduling

In data centers, it is important to detect elephant flows with high traffic demands and properly route them to minimize network congestion. With HONE, we can easily build an application to schedule the elephant flows based on Hedera [3] and Mahout [8]. We implement Hedera's *Global-first-fit* routing strategy with 140 lines

| Management Task | Lines of Code |
|---|---|
| Summing application's throughputs | 70 |
| Monitoring CPU and memory usages | 24 |
| Collecting connection TCP statistics | 19 |
| Calculating traffic matrix | 85 |
| Calculating link utilizations | 48 |
| Discovering network topology | 51 |
| Network performance diagnosis | 56 |
| HONE's directory service | 31 |
| Elephant flow scheduling | 140 |
| Distributed rate limiting | 74 |

Table 6: Traffic-management tasks we build in HONE. Source available at http://hone.cs.princeton.edu/

of code in HONE. The code of the management task has been already shown in previous sections as an example.

We deploy HONE on EC2 instances to emulate a data-center network with a 8-host-10-switch fat-tree topology (the switches are instances running Open vSwitch). We repeat an all-to-all data shuffle of 500MB (i.e., a 28GB shuffle) for 10 times. The HONE-based solution finishes the data shuffle with an average of 82.7s, compared to 103.1s of using ECMP. The improvement over shuffle time is consistent with Hedera's result.

## 5.2 Distributed Rate Limiting

Distributed rate limiting in data centers is used to control the aggregate network bandwidth used by an application, which runs on multiple hosts. It can help the application's owner to control the total cost of using a pay-per-use cloud provider.

Prior works [25, 26] proposed mechanisms to make distributed rate-limiters collaborate as a single, aggregate global limiter inside the network. HONE enables distributed rate limiting from the host side, which introduces less overhead as the hosts have more computational power than the switches, and better visibility into the traffic demand of applications.

In HONE, the administrators do not need to worry about the complexity of collecting throughputs from multiple hosts in a synchronized way. Instead, they just need to write a simple program that sums up throughputs of an application's connections on each host, aggregates the throughputs across hosts, and then calculates their rate-limiting policies accordingly. The code written in HONE are shown below:

```
def DistributedRateLimiting():
  (Select([App, SrcIp, DstIp,
           BytesSent, Timestamp]) *
   From(Connections) *
   Where(App == X) *
   Every(Seconds 1) )                 >>
  ReduceSet(CalculateThroughput, {}) >>
  MapSet(LocalAgg)                       >>
  ReduceSet(MovingAverage, initValue)>>
  MergeHosts()                           >>
  MapStream(GenerateRateLimitPolicy) >>
  RegisterPolicy()
```

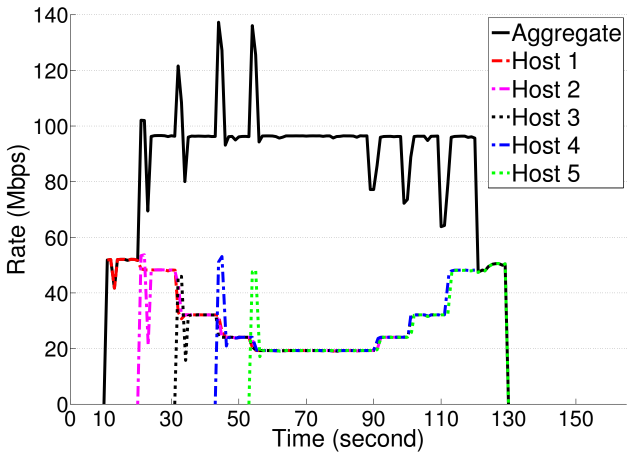


Figure 11: Time series of rates for an application. Every 10 seconds, a host launches the application.

We run the task to limit the aggregate throughput of application $X$ to 100Mbps. The application $X$ is set to send traffic on each host for 80 seconds with a default rate of 50Mbps. We launch $X$ on 5 hosts, one by one every 10 seconds. Figure 11 shows the time series of the aggregate and individual traffic rates of $X$. The management task succeeds in limiting the total rate of $X$ running on a distributed set of hosts to 100Mbps. Note that it takes one round of execution for the management task to discover new traffic and update the rate-limiting policies. That is why there are several 1-second spikes when $X$ starts on new hosts.

## 6. RELATED WORKS

Recent projects have sought to incorporate end hosts into network management [10, 15]. But these solutions view the hosts only as software switches or as trusted execution environments for the network. OpenTCP [13] explores more control APIs specifically over TCP. HONE aims at supporting diverse traffic-management tasks, and thus provides more measurement, data analysis, and control functionalities across hosts and switches.

There have also been industry efforts in simplifying cloud management, such as various commercial tools from vendors [5, 6, 22, 30]. They aim at enabling better visualization and infrastructure control at hosts and switches in the cloud. HONE is complementary to these systems by focusing more on monitoring and analysis in traffic-management tasks and providing programmable interfaces for these tasks.

Prior works also adopt the stream abstraction for network traffic analysis [4, 7]. But they mainly focus on extending the SQL language, while we use functional programming to define traffic-management mechanisms more expressively. Further, some of these works [4, 23] focus on a specific problem (e.g., intrusion detection) when designing their programming language, while HONE aims for a more generic programming interface for traffic management.

Finally, there are recent works proposing network programming languages, such as ProgME [36], Frenetic [12], OpenSketch [35], and Netcalls [27]. They focus on a much narrower class of programming units–raw packets or traffic counters on a single switch, while HONE mainly moves the programmability to the end hosts, aims at an extensible platform for various types of measurement, and encompasses a joint host-network scenario.

## 7. CONCLUSION

HONE is a programmable and scalable platform for joint host-network traffic management in data centers. HONE offers data-center administrators (i) an integrated model of diverse, fine-grained statistics from both hosts and switches and (ii) a simple, expressive, centralized programming framework for defining the measurement, analysis, and control functionality of traffic-management tasks. The programming framework combines a domain-specific query language with a data-parallel analysis framework and a reactive control schema. The system scales through lazy materialization of the measurement data, filtering and aggregating data on each host, and performing hierarchical aggregation of data across multiple hosts. Micro-benchmarks and experiments with real management tasks demonstrate the performance and expressiveness of our system.

In our future work, we plan to build a wider range of management tasks, both to further demonstrate HONE's expressiveness and to continue optimizing our prototype's performance. In addition, we plan to include more support for virtual-machine environment, such as adding VM-level monitoring and migration, and support for more robustness, such as preventing the host agents and the controller from becoming overloaded. We believe that these capabilities, along with our existing support for programmable and scalable traffic management, can make HONE an invaluable platform for data-center administrators.

## References

[1] Event Tracing for Windows. http://support.microsoft.com/kb/2593157.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *ACM SIGCOMM*, 2008.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *USENIX NSDI*, April 2010.

[4] K. Borders, J. Springer, and M. Burnside. Chimera: A Declarative Language for Streaming Network Traffic Analysis. In *USENIX Security*, 2012.

[5] Boundary. http://www.boundary.com/.

[6] Cisco Cloud Management Tools. http://www.cisco.com/en/US/netsol/ns1133/index.html.

[7] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *ACM SIGMOD*, 2003.

[8] A. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-Overhead Datacenter Traffic Management using End-Host-Based Elephant Detection. In *IEEE INFOCOM*, 2011.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI*, 2004.

[10] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. Anderson, and A. Krishnamurthy. ETTM: A Scalable Fault Tolerant Network Manager. In *USENIX NSDI*, 2011.

[11] FloodLight. http://floodlight.openflowhub.org/.

[12] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ACM ICFP*, 2011.

[13] M. Ghobadi, S. H. Yeganeh, and Y. Ganjali. Rethinking End-to-End Congestion Control in Software-Defined Networks. In *ACM HotNets*, Oct. 2012.

[14] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *ACM SIGCOMM*, 2012.

[15] T. Karagiannis, R. Mortier, and A. Rowstron. Network Exception Handlers: Host-network Control in Enterprise Networks. In *ACM SIGCOMM*, 2008.

[16] Linux Advanced Routing & Traffic Control. http://www.lartc.org/.

[17] M. Mathis, J. Heffner, and R. Raghunarayan. TCP Extended Statistics MIB. RFC 4898, May 2007.

[18] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM CCR*, 38(2):69–74, 2008.

[19] Netfilter.org. http://www.netfilter.org/.

[20] NTP: The Network Time Protocol. http://www.ntp.org/.

[21] Open vSwitch. http://openvswitch.org/.

[22] OpenTSDB Project. http://www.opentsdb.net/.

[23] X. Ou, S. Govindavajhala, and A. W. Appel. MulVAL: A Logic-based Network Security Analyzer. In *USENIX Security*, 2005.

[24] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker. Extending networking into the virtualization layer. In *ACM HotNets*, Oct. 2009.

[25] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the Network in Cloud Computing. In *ACM SIGCOMM*, 2012.

[26] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud Control with Distributed Rate Limiting. In *ACM SIGCOMM*, 2007.

[27] J. Sherry, D. C. Kim, S. S. Mahalingam, A. Tang, S. Wang, and S. Ratnasamy. Netcalls: End Host Function Calls to Network Traffic Processing Services. Technical Report UCB/EECS-2012-175, U.C. Berkeley, 2012.

[28] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the Data Center Network. In *USENIX NSDI*, 2011.

[29] R. van Renesse and A. Bozdog. Willow: DHT, Aggregation, and Publish/Subscribe in One Protocol. In *IPTPS*, 2004.

[30] VMWare vCenter Suite. http://www.vmware.com/products/datacenter-virtualization/vcenter-operations-management/overview.html.

[31] Web10G Project. http://web10g.org/.

[32] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *ACM SIGCOMM*, 2011.

[33] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *ACM SIGCOMM*, 2004.

[34] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling Network Performance for Multi-tier Data Center Applications. In *USENIX NSDI*, 2011.

[35] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *USENIX NSDI*, April 2013.

[36] L. Yuan, C.-N. Chuah, and P. Mohapatra. ProgME: Towards Programmable Network Measurement. In *ACM SIGCOMM*, August 2007.

[37] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *ACM SIGCOMM*, 2012.