Edison: Event-driven Distributed System of Network Measurement

Xiaoban Wu†, Timothy Miskell†, Yan Luo*
Department of Electrical and Computer Engineering
University of Massachusetts Lowell
*{Yan_Luo}@uml.edu,
†{Xiaoban_Wu, Timothy_Miskell}@student.uml.edu

Liang-Min Wang‡, Li-De Chen‡
Intel Corporation
Hudson, MA 01749
‡{Liang − Min.Wang, Li − De.Chen}@intel.com

***Abstract*—Network measurement is critical in network management such as performance monitoring, diagnosis, and traffic engineering. However, conventional network measurement software tools are inadequate in high-speed network at scale. They also lack an event-driven mechanism to automate complex network measurement process. In this paper, we present *Edison* which contains an high performance measurement backend (Ares) to collect flow metrics, driven with an expressive frontend (Equery) to enable the composition of complex measurement tasks. Finally, we evaluate the effectiveness of our *Edison* framework on a distributed measurement platform deployed for 100Gbps international research networks.**

***Index Terms*—Network Management; Event-driven Programming; Distributed Network Measurement;**

I. Introduction

Network measurement is instrumental to understanding network behavior, fault diagnosing, attack prevention and traffic engineering. While conventional measurement tools such as `pmacct, argus`[1], [2] support basic network flow metrics (e.g. NetFlow, sFlow), emerging network infrastructure and applications impose new challenges that make these tools out of date.

First, the existing tools were designed a decade ago for the network infrastructure of that time. The computing and networking technology have experienced significantly advance in performance. In particular, the bandwidth of the networks has been increased dramatically from the scale of Mbps to 100 Gbps and beyond. However, the conventional measurement tools do not scale naturally with computing platform and network infrastructure due to inherent design principles and assumptions. For example, our experiment shows that `argus` and `pmacct` can only sustain up to 10-20 Gbps input traffic due to their lack of support to emerging multicore architecture and high performance network I/O.

Second, the new network architecture and operational methods introduce many interesting opportunities as well as new problems. For example, a network flow going through multiple AS domains may experience performance loss at one link, as a result, the trouble shooting on such a flow requires multiple measurement actions to take place, at different locations and in a certain order, before finally narrowing down the problematic point. Although some support SQL-like language to let user choose measurement fields, the conventional measurement tools (e.g. `argus` and `pmacct`) lack a mechanism to express the logical and temporal order of such chained network measurement tasks. In general, a meaningful network measurement is composed of a collection of metric oriented tasks, and network operators execute them one after another. We define an *event* as the trigger that initiates a new measurement task after the prior one completes. We argue that such *events* play an increasingly important role in chaining the tasks to fulfill the desired measurement goals. We can significantly improve the efficiency of network measurements by using *events* as the glue logic to automate the process and avoid human intervention.

To address the aforementioned challenges, we present `Edison`, an event-driven distributed system of network measurement, which includes a high performance backend `Ares` for network metrics collection and a flexible frontend `Equery` for expressive task definition. Specifically, `Ares` is a network measurement component for collecting a variety of flow statistics. It takes advantage of a fast packet I/O library called DPDK [3], an effective parallel design for scalability in multicore systems, and an efficient insertion-lookup data structure called cuckoo hash. Our experiment results show that `Ares` can reach line rate with 100Gbs input traffic. Second, `Equery` is an event-driven query language, which not only allows human operators to launch network queries to derive network-wide information in a programmable way, but also supports event-driven mechanism to detect specified network events and trigger further network monitoring automatically. We demonstrate through extensive evaluation experiments that the `Edison` framework is scalable and effective.

II. Design of Edison

A. Overview of Edison

The design of `Edison` originated from the need of a programmable network measurement framework for International Research Network Connections (IRNC), a global

978-3-903176-15-7

scale high speed network infrastructure for data-intensive scientific research funded by the US National Science Foundation [4]. **Edison** aims to provide advanced passive traffic measurement with flow granularity. By design, **Edison** consists of a growing number of passive measurement instrument deployed at major network exchange points, as show on the map in Figure 1. Each instrument is built upon a multicore x86 server with 100Gbps network interface cards (NICs), and runs **Ares**, the backend for flow metrics collection. The user-facing part of **Edison** is **Equery**, a SQL-like event driven language that allows users to compose complex measurement functions. **Ares** communicates with **Equery** via a central **Edison** controller.

The **Edison** architecture is currently deployed at multiple operational exchange points including StarLight [5] at Northwestern University, University of Kentucky, and AMPATH [6] at Florida International University. Two additional instrument are being deployed and tested at Sao Paulo, Brazil and La Serena, Chile. Other network exchange points are in the planning stage. At each operational exchange point, in order to passively collect flow statistics, the **Ares** runs on a multicore server connected to a mirroring port of a network switch. As solely a passive measurement, **Ares** is different from the active measurement systems like perfSONAR [7]. PerfSONAR actively generates traffic to exercise the network links and collect metrics, whereas **Ares** only passively listens to live traffic to collect metrics, thus having no performance impact to live traffic.

In order to fully take advantage of **Ares**, the **Equery** is designed to flexibly express the event-driven measurement tasks. Users' query statements are analyzed and compiled into small deployable tasks. The **Edison** controller maps these tasks to the distributed **Ares** instrument boxes, which actually execute them. The measurement results are returned to the controller and used to trigger the next task if the event criterion is met. The aggregated results are recorded and eventually made available to user through a web interface.

Next we introduce in detail how **Ares** is designed to support deep programmability and scale with high-throughput network traffic. Then we will describe how **Equery** facilitates composition of complex network measurements.

B. *Edison* Backend: The *Ares*

Ares serves as the backend of **Edison** and runs on a multicore based hardware platform. From the architectural perspective, **Ares** differs from commercial network appliances. **Ares** assumes only the multicore architecture (e.g. x86) and fast packet I/O library (e.g. DPDK [3]), which are readily available from many vendors. It is designed to be open hardware and software, which can evolve well with advancing technologies. Such a salient advantage that has proven to be effective as **Ares** can readily execute on hardware platforms with different configurations. Such

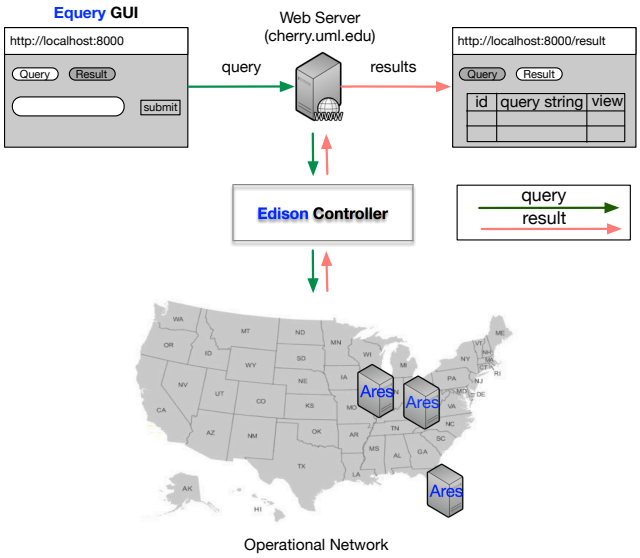


Fig. 1. Edison Architecture

open architecture and familiar programming environment (Linux) also enable rapid prototyping and reprogramming to add and test new measurement functionalities. Both the performance and programmability have made **Ares** the ideal platform for traffic measurement on the research network backbone.

Performing measurement functions at line rate is non-trivial, especially on such a general purpose environment as **Ares**. One challenge is how to best utilize the CPU cores, and the other one is the fast packet I/O. These two challenges are in fact intertwined due to the packet buffer management schemes and workload balancing.

Figure 2 illustrates the overall architecture of **Ares**. From bottom to top, **Ares** has a fast packet I/O scheme, a CPU core management scheme in groups of "clusters", and a software based hash called Cuckoo for efficient look up and insert of flow meta data required for collecting metrics. **Ares** is designed generic enough that these three components are portable and customizable to different hardware platforms. In the next few subsections we explain in more details the specifics of current implementation .

1) Packet I/O and buffer management: It is critical to move received packets from NICs to system memory under tight time constraint for online processing. The per packet packet processing time is only about 10ns for a 64B packet plus padding on wire at 100Gbps. That includes the time needed to bring the packet into the system memory. Unlike the traditional packet capture library *libpcap* [8], used in **pmacct** and **argus**, which works in the kernel space, DPDK is a set of userland libraries and drivers to provide high throughput packet I/O in high speed networks. Recent studies have proven DPDK to be a reliable and feasible framework for high performance software switches on a multicore system. Zhou et al. [9] integrate the DPDK library with a Cuckoo hash table to design a Cuckoo switch operating at significantly high throughput rates.

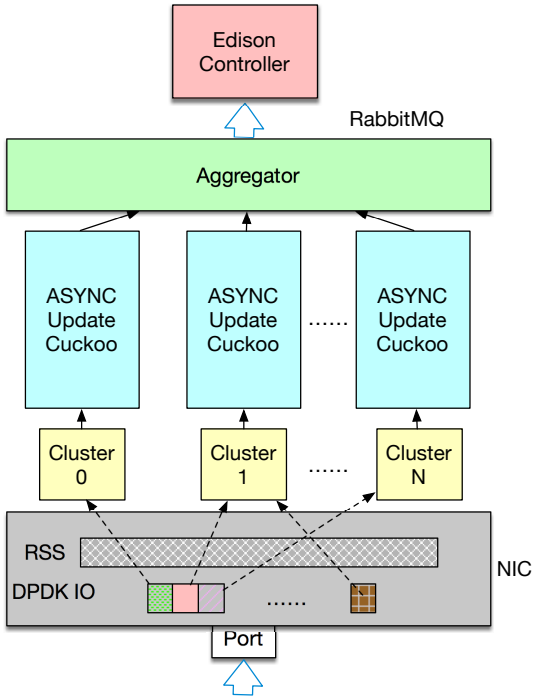


Fig. 2. Ares Architecture

Furthermore, Li et al. [10] propose several VNF designs that aim to increase the performance of network functions by utilizing the DPDK library.

We wish to explicitly emphasize three aspects of DPDK that are most relevant to `Ares`, multicore framework, poll-mode drivers (PMDs) as well as ring buffers.

Multicore Framework: To support Non-Uniform Memory Access (NUMA) architecture which is widely adopted on modern high-end servers, DPDK provides a multicore framework. At the Environment Abstraction Layer (EAL) phase, DPDK detects the enabled cores in the system and manages the software thread affinity to hardware cores. In order to reduce the memory access delay, DPDK enables threads to allocate hugepage memory from the closest socket.

PMD Driver: The PMD driver in DPDK replaces traditional interrupt-driven hardware events with a low latency polling mechanism. The PMD driver resides in the userland space to avoid the overhead associated with user-to-kernel context switches, and this is main reason why `Ares` can sustain high throughput while `pmacct` and `argus` cannot. In addition, the majority of modern NICs support Receive Side Scaling (RSS), which uses hardware hashing to enable the efficient distribution of network receiver-side processing across the multiple CPUs available within a multicore system. As shown in Figure 2, to fully explore the parallelism available within a multicore system, `Ares` declares multiple receiving (RX) queues at the same time and utilizes RSS to distribute packets into multiple RX queues which are further processed by the corresponding Cluster.

Ring Buffer: In order to make communication between threads convenient and efficient, DPDK provides four types of ring buffers implementations for different use case scenarios: single-producer (SP), single-consumer (SC), multiple-producer (MP) and multiple-consumer (MC). The MP and MC ring buffers consist of atomic index updating and parallel reading and writing operations. By default, all the RX queues use MP and MC ring buffers. Since atomic operations are very expensive, using MP and MC buffers may not render the optimal performance. Due to this fact, as explained in section II-B2, within each Cluster, `Ares` utilizes SP and SC ring buffer to implement communication between master core (producer) and slave cores (consumers).

2) CPU core management and load balancing: The number of cores in a x86 server keeps increasing. The latest generation consists of 28 cores in a processor[11], and there are typically two processors or more in a system. It remains an open question as how to allocate cores for high speed network applications. As packet I/O requires polling and active buffer management, certain cores are dedicated for such tasks. The rest of the cores in `Ares` should be utilized to their full extent for flow metrics collection.

In order to efficiently collect flow statistics at line rate, as shown in Algorithm 1, `Ares` has two ways to utilize the CPU cores, namely Aggregator as readers and Cluster as writers. While Aggregator is used to collect flow statistics and report them to `Edison` controller (the communication is done via a software library called RabbitMQ), Cluster is the main component of `Ares` and used to generate flow statistics. In this paper, we only focus on the detail of the design of the writers, i.e. Cluster, which includes a single master core and multiple slave cores.

Algorithm 1 Algorithm of `Ares`

```
1: function Cluster_Subroutine( )
2:     while True do
3:         Asynchronously update cuckoo hash table;
4: function Aggregator_Subroutine( )
5:     while True do
6:         Iterate through each cuckoo hash table
7:                 and collect flow statistics;
8:         Return collected information to Edison controller;
```

Due to the fact that the number of RSS hardware RX queues is limited while a multicore server can have tens of cores, to take the full advantage of these cores and make performance scalable, `Ares` first splits all the cores into multiple Clusters. And then within each Cluster, as shown in Figure 3, `Ares` implements an asynchronous parallel design called ASYNC. ASYNC means that the master core and all the slave cores are working in an asynchronous way and do not wait for any synchronizing signal. We use Figure 3 to depict this: the master core maintains a global ring buffer for each slave core. Whenever the master core receives a batch of packets into RX Queue from NIC via RSS, it hashes each packet and push it into

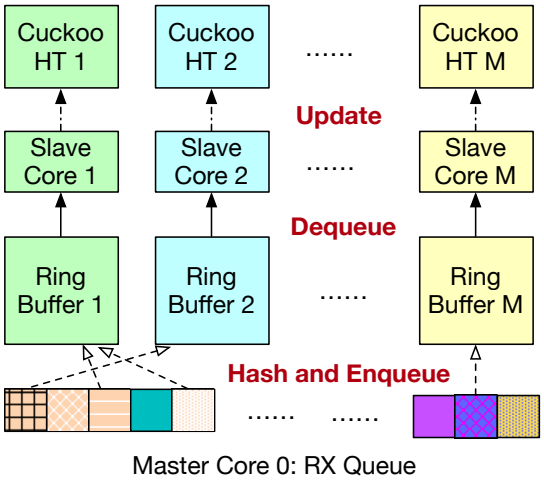


Fig. 3. ASYNC Parallel Design

the corresponding ring buffer of the slave core. For each slave core, it keeps polling on its associated ring buffer and begins to work on the packets immediately if there exists any. The details are shown in the Algorithm 2.

Algorithm 2 Cluster_Subroutine

```
1: Global Data: slave_ring_buffer[]
2: DPDK API: rx_burst(), enqueue(), dequeue()
3: function Master_Core_Subroutine( )
4:     while True do
5:         rx_burst(received_pkts[]);
6:         for each packet in received_pkts[] do
7:             value = hash(packet);
8:             enqueue packet into slave_ring_buffer[value];
9: function Slave_Core_Subroutine( )
10:    while True do
11:        while my buffer in slave_ring_buffer[] has packets do
12:            dequeue(a batch of packets out of my buffer);
13:            Update my cuckoo hash table with dequeued packets;
```

In summary, by using both DPDK RSS (hardware hashing) and ASYNC (software hashing), `Ares` can fully distribute packets into many cores, balance the work load across multiple slave cores and thus achieve the highest scalable throughput.

3) Cuckoo Hash Table for Efficient Lookup and Insertion: To collect flow-level metrics, a measurement function needs to frequently access the per flow stateful meta data (e.g. the total number and size of packets). Therefore, the lookup and insertion operations on such shared stateful data structures are time critical. The cuckoo hash table achieves high memory utilization efficiency, and ensures expected O(1) lookup time. `Ares` adopts 1,8 cuckoo hash table, where 1 means one hash table and 8 means 8-way associative, simply each bucket (row) contains 8 slots. Cuckoo hash table maintains two hash functions, namely primary and secondary hash function. And each hash function map the key to the corresponding bucket, namely primary and secondary bucket. When lookup, cuckoo hash table only needs to check if the key exists in any slot of either primary bucket or secondary bucket. And when insertion of a new key, it first looks for any available empty slot in the primary bucket or the associated set of secondary buckets. If not, it recursively relocate the first unmarked slot from the primary bucket to its secondary bucket until an empty slot is found.

For the *key−value* pair in the cuckoo hash table, `Ares` uses 5-tuple, src_addr, dst_addr, protocol, src_port, and dst_port as *key*, which uniquely defines a flow. And the *value* includes node_id, stime, ltime, smac, dmac, ip_ttl, ip_tos, tcp_seq, tcp_ack, tcp_win, total_count, total_byte, ploss_count, and ploss_byte.

C. *Edison* Frontend: The *Equery*

`Equery` serves as the frontend of `Edison`. The Equery language is not only a SQL-like query language which allows user to gather per-flow statistics information, but also has its own unique features which supports the event-driven mechanism. With an *event*, a user can construct a composite query which contains multiple atomic queries, one triggered by another, and thus avoid manual intervention.

TABLE I
THE EQUERY SYNTAX

Name	Format and Attribute
Atomic Equery	*qName*: **select** *selField* **groupby** *hdrField* **where** *whereCond* **when** *whenCond*;
Composite Equery	a list of Atomic Equery
selField	hdrField aggrField
hdrField	`Ares` measurement fields from *key−value* pair in cuckoo hash table
aggrField	count(hdrField) sum(hdrField) avg(hdrField) max(hdrField) min(hdrField)
whereCond	boolean expression of hdrField
whenCond	boolean expression of whenField
whenField	*qName.selField*

1) The Equery Syntax: The syntax of `Equery` is summarized in Table I. For more details of `Equery`, we refer readers to [12].

A **select** clause is used to express what data the user wants to query. Here the arguments are defined as measurement fields, including the fields of a packets header, as well as the aggregated fields.

A **where** clause applies the boolean expression to filter the query results. Only those packets satisfying the boolean expression will be analyzed for statistical purposes.

A **groupby** clause is used together with an **select** clause to collect measurement data across multiple fields and group the results by one or more columns.

A **when** clause is employed to impose a *event* trigger on a query. A query driven by a **when** clause will be executed

once the *event* (described as boolean expression) happens. With **when** clause, `Equery` uniquely distinguishes it self from the conventional measurement tools, for example, `pmacct` and `argus`.

2) The `Equery` Examples: We present two `Equery` examples to illustrate how to use `Equery` to describe measurement tasks. The first one is an atomic `Equery`, and the second one is an event-driven composite `Equery`.

Example 1: Atomic Equery

- q1: **select** count(src_addr), dst_addr **groupby** dst_addr **where** node_id=NodeX;

This example can be used to count the number of sources for each destination, and hence detect the possible DDOS attack at NodeX.

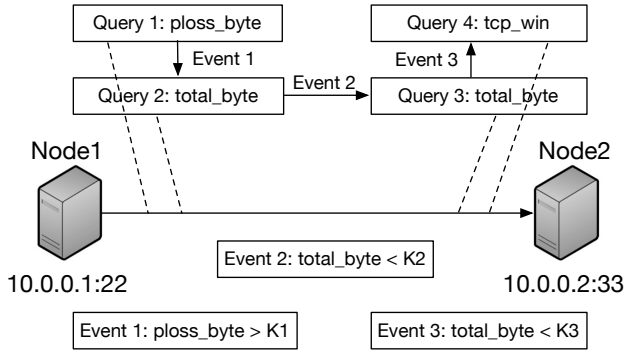


Fig. 4. Example network for event mechanism.

Example 2: Composity Equery

As shown in Fig. 4, in order to detect packet loss between two Data Transfer Nodes (DTN) and the root causes along the path from Host A to Host B, we can describe the process as follows.

- q1: **select** ploss_byte **where** src_addr=10.0.0.1, dst_addr=10.0.0.2, src_port=22, dst_port=33, protocol=TCP, node_id=Node1;
- q2: **select** total_byte **where** src_addr=10.0.0.1, dst_addr=10.0.0.2, src_port=22, dst_port=33, protocol=TCP, node_id=Node1 **when** q1.ploss_byte > K1;
- q3: **select** total_byte **where** src_addr=10.0.0.1, dst_addr=10.0.0.2, src_port=22, dst_port=33, protocol=TCP, node_id=Node2 **when** q2.total_byte < K2;
- q4: **select** tcp_win **where** src_addr=10.0.0.1, dst_addr=10.0.0.2, src_port=22, dst_port=33, protocol=TCP, node_id=Node2 **when** q3.total_byte < K3;

After the query q4, we check to see if the TCP window size is the root cause of packet loss. If so, we can start to tune the relevant parameters of system settings in the DTNs.

III. Evaluation

In this section, we evaluate the performance of `Edison`. In particular we focus on the performance and scalability of `Ares` as `Equery` has been used in experimental operations of the international research networks. We expect to show the performance boost of `Ares` over `argus` and `pmacct` on measuring high speed network traffic up to 100Gbps. The evaluation metric is packet receiving rate (PRR), which is proportional to the sustained throughput.

A. Experiment Platform and Methodology

We employ two identical Dell Power Edge R730 servers in the experiments. Each server contains two Intel Xeon E5-2643 6-core CPUs @ 3.4GHz that reside separately on 2 sockets (socket #0 and #1) connected with Intel QuickPath Interface (QPI). Both servers are equipped with a single Mellanox ConnectX-4 EDR 100GbE NIC on the socket #1 over a PCIe x16 slot. The aforementioned two NICs are connected back-to-back via a QSFP-28 cable for testing purpose. We set one server as the TX server for packet generation and the other server as the RX client for the `Ares` performance testing.

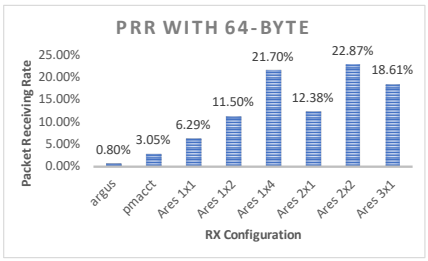
B. TX Capability Study

To emulate the realistic network traffic at operational exchange point, which might involve many different flows at the same time, we have modified DPDK *pktgen* [13] to generate random flows at the highest rate. For each of the following RX evaluation, our modified pktgen always sends out 1.5M different flows. The TX capability varies with packet sizes: it reaches 55Gbps for 64-byte packets, and 94Gbps for 750-byte packets. The packet rate (million packets per second or Mpps) is 78 Mpps for 64-byte ones, and 16 Mpps for 750-byte ones, respectively. It is expected that the smaller the packets (e.g. 64-byte), the more challenging for the `Ares` backend to process due to per packet I/O overhead.

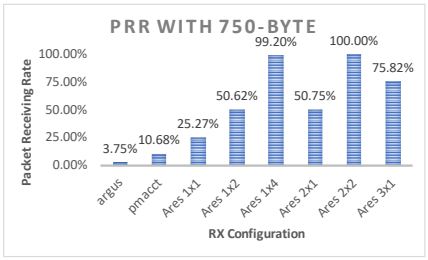
C. RX Capability Study

As shown in Figure 5, we compare the packet receiving rate between different RX configurations. For `Ares`, NxM means it has N Clusters and each Cluster has M slave cores, whereas `argus` and `pmacct` have no such setting. Also since there are only 6 CPU cores at the socket where the NIC is inserted, we can only have up to 3 Clusters. With these experiment data, we have some observations as follows.

- The `Ares` has huge performance boost against `argus` and `pmacct`. With `Ares` 2x2 configuration, the PRR is around 9x better than `argus` and `pmacct`. This comparison clearly shows that our design of `Ares` is efficient under 100 Gbps network environment.
- The `Ares` has good performance scalability. For example, with the same number of Clusters, either 1x1, 1x2 and 1x4, or 2x1 and 2x2, the PRR scales linearly with the number of slave cores. And with different number of Clusters, 1x1, 2x1 and 3x1, the PRR also scales linearly. These data proves that `Ares` can scale very well in a multicore system.



(a) 64-byte



(b) 750-byte

Fig. 5. Packet Receiving Rate with Different Packet Sizes

- Between either 1x2 and 2x1, or 1x4 and 2x2, they have the same number of slave cores, but PRR is always slightly better when more Clusters are used. This is due to the fact that the overhead of hardware hashing is always less than software hashing. This means we should always prioritize the number of Clusters if we want to maximize the PRR.

IV. Related Work

There are several prior research works thar are closely related to ours, and they are mainly separated into two categories based upon their functionalities, backend and frontend. **1) Backend** Lucente developed `Pmacct`[1], a small set of multi-purpose passive network monitoring tools. Bullard developed `Argus`[2], an IP auditing tool to help support network security management and network forensics. Fusco et.al[14] developed packet capture kernel module that enables monitoring applications to scale with the number of cores. However, these tools do not provide adequate performance for high-speed networks. **2) Frontend** Foster *et al.* designed Frenetic [15], a high-level declarative query language for programming distributed collections of network switches. Narayana *et al.* [16] presented Marple, a declarative query language that targets P4-programmable software switch. However these languages do not provide a mechanism to describe flow-level events which allow for triggering subsequent queries.

V. Conclusion and Future Work

In this paper we presented `Edison`, an event-driven distributed system of network measurement. `Edison` has not only a powerful backend `Ares` which can sustain 100 Gbps traffic, but also a flexible frontend `Equery` which enables event-driven measurement. Our evaluation shows that `Edison` is a significant improvement over the existing traditional network measurement tools, in both backend and frontend. In the future, we will deploy `Edison` into more complex networks and construct more use cases.

Acknowledgment

This work is supported in part by the National Science Foundation (No. 1547428, No. 1541434, No. 1738965 and No. 1450996), Hubei ISTC Program (2017AHB060) and a grant from Intel Corporation.

References

[1] P. Lucente, "pmacct: steps forward interface counters," *Tech. Rep.*, 2008. [Online]. Available: http://www.pmacct.net
[2] L. QoSient, "Argus: Auditing network activity," 2009. [Online]. Available: https://qosient.com/argus
[3] Intel Corporation, "Data plane development kit," 2018. [Online]. Available: http://dpdk.org
[4] K. Thompson and D. Gatchell, "Nsf international research network connections (irnc) program," in *NSF IRNC Program Kickoff Meeting*, 2005.
[5] J. Mambretti, T. DeFanti, and M. D. Brown, "Starlight: Next-generation communication services, exchanges, and global facilities," in *Advances in Computers*. Elsevier, 2010, vol. 80, pp. 191–207.
[6] "Ampath: The international exchange point for research and education networking in miami." [Online]. Available: https://www.ampath.net
[7] B. Tierney, J. Metzger, J. Boote, E. Boyd, A. Brown, R. Carlson, M. Zekauskas, J. Zurawski, M. Swany, and M. Grigoriev, "perfsonar: Instantiating a global network measurement framework," *SOSP Wksp. Real Overlays and Distrib. Sys*, 2009.
[8] L. M. Garcia, "Programming with libpcap-sniffing the network from our own application," *Hakin9-Computer Security Magazine*, vol. 2, p. 2008, 2008.
[9] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance ethernet forwarding with cuckooswitch," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 2013, pp. 97–108.
[10] P. Li, X. Wu, Y. Ran, and Y. Luo, "Designing virtual network functions for 100 gbe network using multicore processors," in *Architectures for Networking and Communications Systems (ANCS), 2017 ACM/IEEE Symposium on*. IEEE, 2017, pp. 49–59.
[11] Intel Corporation, "Intel xeon platinum 8180m processor," 2018. [Online]. Available: https://ark.intel.com/products/codename/37572/Skylake
[12] Y. Ran, X. Wu, P. Li, C. Xu, Y. Luo, and L. Wang, "Equery: Enable event-driven declarative queries in programmable network measurement," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, April 2018, pp. 1–7.
[13] W. Keith, "The pktgen-dpdk packet generator," 2018. [Online]. Available: http://dpdk.org/browse/apps/pktgen-dpdk
[14] F. Fusco and L. Deri, "High speed network traffic analysis with commodity multi-core systems," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 218–224.
[15] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '11. New York, NY, USA: ACM, 2011, pp. 279–291. [Online]. Available: http://doi.acm.org/10.1145/2034773.2034812
[16] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 85–98.