MEASUREMENT AND MONITORING FRAMEWORK FOR THE EVOLVING INTERNET

by

NOLAN RUDOLPH

DISSERTATION ABSTRACT

Nolan Rudolph

Bachelor of Science

Department of Computer and Information Sciences

June 2020

Title: Measurement and Monitoring Framework for the Evolving Internet

Granted the annual trends in increasing internet usage, the demand for scalable and high-performance networks continues to rise. Furthermore, as networks have evolved in new and profound ways, the tools and practices used to measure them have not always kept pace with their evolution. In light of both these events, this paper aims to provide a deeper insight on the measurement and monitoring frameworks for the evolving Internet and the design they should endow. We begin by producing a light-weight and scalable network flow to packet synthesizer designed to be a telemetry system for new measurement frameworks. Next, we examine the Extended Berkely Packet Filter and its applicability to active and passive network measurements by designing an Express Data Path flow collector. Lastly, we present MicroMon, a multi-dimensional monitoring framework for geo-distributed applications using heterogeneous hardware and receive 10-50% throughput gains when applied to third party software.

CURRICULUM VITAE

NAME OF AUTHOR: Nolan Rudolph

UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA

DEGREES AWARDED:

Bachelor of Science, Computer and Information Sciences, 2020, University of Oregon
Secondary Academic Discipline, Mathematics, 2020, University of Oregon

AREAS OF SPECIAL INTEREST:

Network Measurements
Software Engineering

PROFESSIONAL EXPERIENCE:

Researcher, Oregon Networking Research Group, Winter 2019 - Summer 2020
Research for Vice President for Research and Innovation Fellowship, University of Oregon's Undergraduate Research Opportunities Program, Summer 2019
Computer Science and Mathematics Tutor, University of Oregon, Winter 2018 - Fall 2020

GRANTS, AWARDS AND HONORS:

VPRI Fellowship, University of Oregon, 2019

PUBLICATIONS:

Durairajan, R., & Kannan, S. (2020). A Monitoring Framework for Tackling Distributed Heterogeneity. *Usenix HotStorage*

(2020). New Capabilities for Self-Driving Networks. *University of Oregon Scholars' Bank*

(2019). Trace Driven Traffic Generator for Self-Driving Networks. *University of Oregon Scholars' Bank*

ACKNOWLEDGEMENTS

First and foremost, I want to thank my mentor, Ramakrishnan Durairajan, for helping me through the entirety of my research and being a significant part of my preparation for the working world. Ramakrishnan has helped me through the highs and lows of all projects, and has taught me a tremendous amount of knowledge regarding his specialized field of study, computer networking. Lastly, I'd be remiss if not to thank the University of Oregon for the teachings they have provided me. From basic Python programming to the intricacies of operating systems, they have presented me with the knowledge required to indulge in the work seen in this thesis as well as future occupational projects.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 2

II. BACKGROUND AND MOTIVATION 4

Netflow to Packet Synthesizer 4

Network Packet Background 4

Program Optimization 4

Harnessing eBPF for Network Measurement 5

eBPF Overview 6

Passive Flow Collector 9

Micro-Metric Network Monitor for Distributed Heterogeneity . . . 9

Programmable Switches and Network Telemetry 10

Growing Resource Heterogeneity and Application Requirements 10

Lack of Multi-dimensional Monitoring 11

III. MEASUREMENT TECHNIQUES 12

FlowSynth 12

Implementation 12

Evaluation 14

eBPFlow 16

Benefits 16

Implementation 17

User Space 17

Chapter Page

Kernel Space 17

Challenges 18

Evaluation 18

Configuration 18

Results 18

IV. MONITORING FRAMEWORK 20

Implementation of MicroMon 20

Micrometrics Selection 20

Micrometrics Collection & Dissemination 21

Scalable Inference Logic 23

Experimental Evaluation 23

Impact of Heterogeneous Storage 24

Network Latency and Page Cache 25

V. CONCLUSION 27

Summary of Results 27

FlowSynth 27

eBPFlow 27

MicroMon 27

Lessons Learned 28

FlowSynth 28

eBPFlow 28

MicroMon 28

Future Work 29

Chapter Page

FlowSynth 29

eBPFlow 29

MicroMon 29

REFERENCES CITED 30

LIST OF FIGURES

Figure | Page

1. Diagram showing primary locations where eBPF programs can be executed (XDP and `tc`), as well as how a user-space control program (via bcc) interacts with eBPF programs and in-kernel data storage (BPF maps). 7

2. Netflow to Packet Synthesizer's performance in PPS, run on Intel Xeon D-1548 2.0 GHz core, MLNX X-3 10 Gb/s NIC. . . . 15

3. TCPReplay to Packet Synthesizer's performance in PPS, run on Intel Xeon D-1548 2.0 GHz core, MLNX X-3 10 Gb/s NIC. 16

4. Maximum sustainable packet rate (Mpps) by eBPFlow for an increasing number of CPU cores. 19

5. **High-level MicroMon Design.** Figure shows the integration of our HW and SW micro-metrics collection and dissemination in MicroMon. 22

6. **Storage Hardware Impact.** Results for YCSB workloads varying the number of threads and the storage hardware across replicas. 24

7. **Network and Storage Latency Impact.** Results show the combined considering network and storage latency for YCSB workload A with 16-threads. 25

CHAPTER I
INTRODUCTION

As the usage of Internet-able devices continues to increase each successive year *ICT Statistics of 2001-2018* (2019); *Internet Growth Statistics 1995 to 2019* (2020); *The Rise of Social Media* (2020), so does the demand for secure, scalable, and high-performance networks. With this requirement remaining pertinent for today's networks, researchers perpetually strive to create new and improved networking systems that will accommodate for the inflation of Internet-able users and the demands of new technology. To evaluate the performance of their services, researchers use passive and active measurements to do so. Passive measurements include methods such as packet tracing and flow collection to gain insight into protocol behavior, to engineer traffic flows, and to detect network intruders. Active (probe-based) measurements are used to provoke certain behaviors out of network protocols, devices, and applications in order to infer or directly assess end-to-end performance metrics, forwarding paths, configurations, and many other characteristics, e.g., Crovella and Krishnamurthy (2006); Jacobson (1997); Paxson, Mahdavi, Adams, and Mathis (1998); Pelsser, Cittadini, Vissicchio, and Bush (2013); Sundaresan, Deng, Feng, Lee, and Dhamdhere (2017). In many instances, these services are deployed over a variety of geographically-distributed data centers with heterogeneous storage, WAN resources, and other non-unanimous variables.

As networking services have continued to grow stronger in terms of throughput and number of capabilities, so has the demand for new and improved telemetry systems with the bandwidth and capacity necessary to allow the proprietor an effective test of their system's efficiency and accuracy. Current telemetry systems lack the ability to accurately capture the entirety of a system's benchmark and leaves researchers skeptical of the results they receive. Considering the deployment of services with configurations that differ geographically and mechanically, though the heterogeneity can be beneficial (e.g. providing DPDK with a compatible NIC), there is a fundamental disconnect between the requirements of geo-distributed applications using heterogeneous resources and today's coarse-grained monitoring frameworks.

To accommodate for the necessary requirements of improved passive and active measurement tools, a number of approaches have been taken. In an attempt to reduce interrupt load and overheads of kernel-/user-mode context switches, the use of *zero-copy buffering* has been exploited by `PF_RING` and `Netmap` *Ntop's PF_RING ZC (Zero Copy)* (n.d.); Rizzo (2012). In a more extreme approach, such as the likes of DPDK *Data Plane Development Kit* (n.d.), frameworks utilize *kernel-bypass* in which a unique kernel-level driver allows the packet direct access to the user-mode program, avoiding the packet processing pipeline in the OS kernel. Although these frameworks satisfy the requirements prompted by the new innovations in

networking services, these frameworks often incur unexpected overhead due to the re-injection of packets into kernel space for sending. Furthermore, these services require special hardware features which limit the number of potential use cases they can be deployed in.

We consider these challenges prompted by the improved efforts of networks, namely the lack of passive/active measurement tools and compatibility of heterogeneous resources with current coarse-grained monitoring frameworks, and posit solutions in the form of two projects, respectively: `eBPFlow` and `MicroMon`. `eBPFlow` utilizes the services of *Extended Berkeley Packet Filter (eBPF)* which offers a way to advance the practice of network measurement by defining a virtual machine instruction set solely for packet filtration. Entailed by eBPF's components is the eXpress Data Path (XDP), a driver-space program that runs within the kernel and integrates cooperatively with the regular networking stack making it much more compatible with a variety of hardware than its counterparts. Due to its low level involvement with the system, which operates entirely within packet-space, no socket buffer is required which emphasizes the importance of using XDP as a passive measurement tool. Regarding an updated heterogeneous resource monitoring framework, we present `MicroMon`, a monitoring framework that efficiently collects, disseminates, and processes the combined impact of storage and WAN heterogeneity – i.e., heterogeneous hardware and software resources – for improving the performance of third party software such as Apache Cassandra *Apache Cassandra* (n.d.). To overcome the problem of coarse-grained monitoring, MicroMon introduces *micrometrics*, which is a set of fine-grained hardware and software metrics required to study the combined impact of heterogeneous resources on application performance.

In this paper, we describe the intricacies of `eBPFlow` and `MicroMon`, identifying their implementations as well as providing an in-depth evaluation of their services within a controlled setting. We also develop an intermediate telemetry tool, `FlowSynth`, designed to synthesize network flows into packets for the testing of our proposed solutions. We observe that `eBPFlow` is able to achieve a maximum packet capture rate of 20 Mpps while monitoring approximately 800K concurrent flows per second, using only 5 cores. We make sure to emphasize the importance of scalability and conduct elaborate tests to ensure this trait is inherited by `eBPFlow`. Lastly, we show 10-50% throughput gains in our usage of `MicroMon` versus the default services provided by Apache Cassandra.

CHAPTER II
BACKGROUND AND MOTIVATION

Netflow to Packet Synthesizer

Network Packet Background. Before coding, we required knowledge on the architecture of network packets as we had no prior experience in network programming. Luckily, Ross's Computer Networking textbook Kurose (2017) was all we needed to get our project underway. In this text, Ross speaks of the OSI model of networking, a seven layer depiction of how a packet can be constructed and transmitted along different mediums and then deconstructed at alternate endpoints in order to efficiently transfer data. Fortunately, Ross talks in depth about the application, transport, network, and link layers which perfectly coincided with the intentions of our project.

1. Link Layer: Once a packet is put on a medium (e.g. fiber optic cables), the link layer is comprised of switches which utilize data frames from packets to forward data to the correct location.

2. Network Layer: Similar to the Link Layer, the Network Layer utilizes a particular Internet protocol specified by the packet to guide the data to the appropriate location.

3. Transport Layer: Here, the method of transportation specified by the packet is used to forward the data in a particular way, be it maintaining a continuous connection between two endpoints or one-way communications.

4. Application Layer: Lastly, this is where the packet finishes deconstructing and presents the payload, or important data of the packet, to the requesting user.

These layers are referred to as layer 1, layer 2, layer 3, and layer 7, respectively. Therefore, we knew we had to translate flow level data into packets which were comprised of these four different layers.

Program Optimization. Continuing this idea of overall optimization, we looked to generic programming textbooks for guidance. One in particular caught our attention, O'Hallaron's Computer Systems textbook, which brought me great insight into a programmer's perspective of code optimization. In this text we learned numerous development techniques that help decrease the number of cycles a computer has to make to run code:

1. Eliminating Loop Inefficiencies: Direct Memory Accessing (DMA) costs computers a considerable amount of processing power and time. Therefore, if there is any way to remove memory references (e.g. using a temporary local variable), then that should be implemented

2. Reducing Procedure Calls: Not only do function calls utilize DMA, but they also run through a set of instructions that can cost many cycles. It is wise to remove gratuitous function calls from repetitive procedures

3. Loop Unrolling: This can be utilized to accomplish multiple iterations in a single iteration inside a loop, removing the need to continuously reference and write to the same memory addresses

4. Program Parallelism: Enhancing parallelism in ones program allows a new process to begin before the last one finishes. This way, no locks will be encountered in critical sections of code.

We would keep all of these strategies in mind throughout the entirety of writing our program.

Harnessing eBPF for Network Measurement

A packet filter is a kernel agent that provides a way to select desired packets from an incoming stream of network traffic and to discard unwanted packets. Packet filters were initially designed to aid in development and debugging of network applications by exposing network traffic to a user-level program for packet trace analysis and/or network protocol handling Mogul, Rashid, and Accetta (1987). Mogul *et al.* claim that packet filters originated with the Xerox Alto computer Thacker, MacCreight, and Lampson (1979) in 1976, and that the first UNIX implementation of a packet filter was in 1980 Mogul et al. (1987). Key concerns in this early work were (1) efficiency of the in-kernel packet selection process and (2) limiting the interrupt load and the overhead of kernel/user-mode boundary crossings. To address (1), the Mogul *et al.* designed a stack-based language for selecting packets based on a fixed offset Mogul et al. (1987). Similarly, Braden describes an instruction set for a packet filtering program that relied, in turn on a promiscuous network tap (*i.e.*, at the time, the SunOS Network Interface Tap Hess, Safford, and Pooch (1992)) for receiving all network traffic from the interface Braden (1988). Additional techniques such as interrupt batching were explored in Mogul (1990) to reduce context switches and improve packet capture performance.

In 1993, McCanne *et al.* identified several performance bottlenecks with prior efforts and described the Berkeley Packet Filter (BPF), which introduced a new filtering language and RISC-like virtual machine model for the filter processing and a new packet buffering subsystem McCanne and Jacobson (1993). BPF significantly improved performance over prior work and has been a *de facto* standard for quite some time. For example, the well-known `libpcap` and `tcpdump` use BPF-based filtering and attach to a network tap/interface to capture and analyze the network traffic *tcpdump and libpcap* (n.d.). Since then, a number of works have proposed improvements on BPF, *e.g.,* Begel, McCanne, and Graham (1999); Ioannidis, Anagnostakis, Ioannidis, and Keromytis (2002); Wu, Xie, and

Wang (2008, 2011). In 2013, the Extended BPF (eBPF) was proposed, with a new virtual machine model, a vastly expanded instruction set, new kernel hooks, and other new facilities such as persistent in-kernel data structures *eBPF - extended Berkeley Packet Filter* (n.d.). More broadly, the programmability enabled by eBPF resembles other efforts in on-device computation and software-defined networking, *e.g.* Bosshart et al. (2014a); Feamster, Rexford, and Zegura (2013); Kohler, Morris, Chen, Jannotti, and Kaashoek (2000); Tennenhouse, Smith, Sincoskie, Wetherall, and Minden (1997) and programmable network measurement, *e.g.* Jeyakumar, Alizadeh, Geng, Kim, and Mazières (2014); Sommers and Barford (2007); Ziviani, Cardozo, and Gomes (2012).

A number of complementary approaches have been taken for reducing interrupt load and overheads of kernel-/user-mode context switches. For example, coalescing interrupts and "batching" packet arrivals can effectively reduce system load Mogul (1990), although it has some negative side-effects for some types of network measurements Prasad, Jain, and Dovrolis (2004). Another successful approach to limiting overheads of packet capture and filtering is *zero-copy buffering*, meaning that packets that pass a given filter and which are destined to a user-mode application are copied directly into a user-space buffer from the device driver. A system that exemplifies this approach is `PF_RING` Cardigliano, Deri, Gasparakis, and Fusco (2011); Fusco and Deri (2010); *Ntop's PF_RING ZC (Zero Copy)* (n.d.). A more extreme approach is *kernel-bypass*, in which a special kernel-level driver passes packets directly to a user-mode program, avoiding the packet processing pipeline in the OS kernel and overheads of its generality. The U-Net system Von Eicken, Basu, Buch, and Vogels (1995) was an early implementation of this idea, and has many similarities to the Exokernel approach developed around that same time Engler, Kaashoek, and O'Toole Jr (1995); Ganger et al. (2002) as well as the more recent Arrakis operating system Peter et al. (2016). The more recent Netmap Rizzo (2012) and DPDK *Data Plane Development Kit* (n.d.) systems take this approach. As noted above, the eXpress Data Path (XDP) component of eBPF has been described as an in-kernel competitor with DPDK in terms of high-throughput packet performance *BPF and XDP Reference Guide* (n.d.); Høiland-Jørgensen et al. (2018); Karlsson and Töpel (2018).

eBPF Overview. The Extended Berkeley Packet Filter (eBPF) is based on the earlier Berkeley Packet Filter, which was designed as a simple virtual machine instruction set specifically for packet filtering McCanne and Jacobson (1993). The earlier BPF is now being referred to as "classic" BPF to distinguish it from its modern successor *BPF and XDP Reference Guide* (n.d.); Schulist, Borkmann, and Starovoitov (n.d.). eBPF first appeared in the Linux 3.18 kernel, which was released in December 2014 *The BPF system call API, version 14* (n.d.); *Linux kernel commit* (n.d.). Development of Linux kernel eBPF facilities has continued at a steady pace; a history of versions and main features added is

maintained by the BPF compiler collection (bcc) project *BPF Features by Linux Kernel Version* (n.d.).

eBPF expands greatly on the notion of an in-kernel virtual machine instruction set. In particular, the VM instruction set has been broadened and generalized for a variety of tasks (*i.e.*, not just packet filtering). eBPF programs are compiled to bytecode using the llvm compiler (with `-march=bpf`). Upon loading into the kernel, programs are verified to ensure that they (1) cannot have invalid memory references and (2) all code paths terminate. Termination is verified by disallowing loops and by evaluating the control flow graph. Program sizes are also limited; in the initial versions of eBPF, programs were restricted to 4096 instructions, but in recent kernels that limit has been raised to 1 million. There are also restrictions on which kernel helper functions can be used, depending on the eBPF program type *BPF and XDP Reference Guide* (n.d.); *A thorough introduction to eBPF* (n.d.). The instruction set resembles that of modern processors, which facilitates just-in-time compilation to the host architecture which is done by default. Currently classic BPF programs *e.g.*, from `tcpdump` are transparently translated to eBPF programs and JIT-compiled, resulting in performance improvements for traditional packet filtering applications *BPF and XDP Reference Guide* (n.d.).

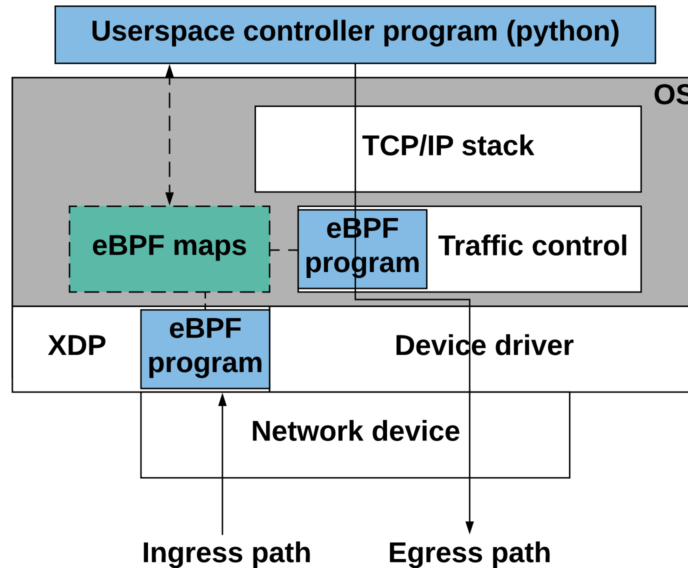

Figure 1. Diagram showing primary locations where eBPF programs can be executed (XDP and `tc`), as well as how a user-space control program (via bcc) interacts with eBPF programs and in-kernel data storage (BPF maps).

eBPF programs are invoked in response to different kernel events. The eBPF *program type* restricts the range of kernel hooks where a program can be installed. For active and passive network measurement, the two most relevant program

types are eXpress Data Path (XDP) Høiland-Jørgensen et al. (2018); *XDP-Project* (n.d.) and Linux's `tc` (traffic control) layer Borkmann (2016); *BPF and XDP Reference Guide* (n.d.); *Linux cls_bpf* (n.d.). The XDP program type can *only* be invoked upon packet ingress; the program is invoked in the device driver *before* any parsing or host processing (device drivers that support this capability are documented by the bcc project *BPF Features by Linux Kernel Version* (n.d.)). `tc` eBPF programs, on the other hand, can be involved either on packet ingress or egress. Clearly this limitation has an impact on the design of eBPF-based active measurement tools. The main differences compared with XDP are that a `tc` program is invoked within the networking stack and operates on a parsed packet (Linux `sk_buff`) *BPF and XDP Reference Guide* (n.d.) whereas XDP programs operate on raw packet contents. For XDP eBPF programs there are facilities for offloading the program to the NIC itself (although Netronome is the only vendor that currently supports this *BPF and XDP Reference Guide* (n.d.)). Figure 1 shows the general architecture of where XDP and `tc` eBPF programs reside in the kernel.

For each of the XDP and `tc` program types, eBPF helper functions facilitate modification of packet contents, recomputing checksums, etc. Helper functions also exist for debug tracing, timestamps, random number generation, and more. The range of helpers that can be used by different program types varies *BPF Features by Linux Kernel Version* (n.d.), but are fairly similar for XDP and `tc` programs. These helper function calls are translated into direct kernel function calls upon JIT compilation.

Since eBPF programs are invoked only in response to packet arrivals (on ingress for XDP or ingress/egress for `tc`), there are additional facilities for persistent data storage and access within the kernel: *BFP maps.* There are a few different types of maps available, including hashmaps, longest prefix match tries, LRU maps, and arrays. In-kernel maps can be accessed from programs running in user mode and maps can be pinned so that they remain resident in the kernel even if the associated eBPF program(s) are not running. Moreover, there are per-cpu variants of some of these data structures, including per-cpu arrays and hashmaps which enable lock-free access to a data structure. Although helpers exist for using spinlocks and atomic addition, spinlocks in particular cannot be held while any other helper function is called, which precludes common synchronization patterns.

There are both generic maps, where the types of keys and values can be user-defined, and more specialized maps. One kind of specialized map is useful for code structuring: the program map (`BPF_MAP_TYPE_PROG_ARRAY`). Program maps contain other eBPF programs, and can be invoked through a special helper function along with the map name and a key. If the key is not in the map, control flow continues in the current program. Otherwise, a longjmp is performed to the new program. Program maps enable some modularization of code designed to handle different protocols, IP versions, etc. *BPF and XDP Reference Guide* (n.d.).

Low-level eBPF programming can be difficult, largely due to the complex API and safety requirements. There are a number of projects to create simplified front-ends for the "raw" eBPF C-based API, including `bpftrace`, `perf`, `ply`, and `bcc` Gregg (n.d.). In our work, we use the bcc (BPF Compiler Collection) tools, which enable creation of a user-space control program in Python (or Lua, Go, Rust and other languages). `bcc` has interfaces for compiling and installing eBPF programs (including XDP and `tc` programs, in conjunction with `pyroute2` *pyroute2 netlink library* (n.d.)), interacting with in-kernel BPF maps, and collecting kernel debug information.

Because of the opportunities afforded by safe, in-kernel packet processing, a number of networking projects leverage eBPF facilities, including flow processing in Open vSwitches Tu (2016), IOVisor project *IO Visor Project* (n.d.); Matteo Bertrone (2016), network virtualization Ahmed, Alizai, and Syed (2018), creation of high-performance P4-based programmable switches *p4 on the edge* (n.d.), network monitoring frameworks *Recap: High-performance Linux Monitoring with eBPF* (n.d.); *Using eBPF for network traffic analysis* (2018), high-performance routing Toonk (2020), techniques for DDoS defense Bertin (2017), networking stack extensibility Bonaventure (2019); Tran and Bonaventure (2019), and routing flexibility Wirtgen (2019); Xhonneux and Bonaventure (2018); Xhonneux, Duchene, and Bonaventure (2018). An excellent compilation of eBPF resources and projects that use eBPF is available on github *A curated list of awesome projects related to eBPF* (n.d.). These projects motivate and mandate a critical rethinking of network measurement tools and practices in the light of benefits offered by eBPF.

Passive Flow Collector. We designed a capability to capture network flows in a cost-effective and scalable manner is the second use case we explore. State-of-the-art approaches either employ dedicated hardware solutions—that are rigid in terms of functionality as well as cost-prohibitive—or deploy an open-source tool to capture network flows that cannot keep up with the traffic at line rate (i.e. limited scalability). On the contrary, an eBPF-based flow collector is poised to turn a commodity Linux device to passively capture network flow with reasonable performance. Furthermore, we posit that such a solution is more flexible and cost-effective compared to hardware solutions.

Micro-Metric Network Monitor for Distributed Heterogeneity

With increasing data processing, analytics, and storage demands, geo-distributed applications are becoming a lifeline of modern enterprises and content providers, spanning across several geo-tropical DCs. These applications range from compute-intensive streaming (e.g., Apache Spark, Hadoop) and batch processing applications to I/O-intensive data serving applications such as NoSQL Cassandra *Apache Cassandra* (n.d.), Google Spanner Corbett et al. (2013),

Amazon's Dynamo that must support millions of operations with microsecond-level latency. In addition, these geo-distributed applications have varying levels of consistency, availability, security, and partition tolerance requirements. For example, compared to streaming applications, data serving applications such as Spanner demand higher availability and stronger consistency; hence data placement and replica selection becomes a key aspect of the application design.

For example, Cassandra allows the end-user request to land on any quorum-based replica node. For data partitioning and request routing across replicas, Cassandra (and other similar applications) use consistent hashing DeCandia et al. (2007). Each node gets assigned to some key range and acts as a coordinator node responsible for replication. The coordinator is responsible for replication of data on different nodes and replicating to other $n - 1$ replica nodes. For replica selection and request routing, Cassandra uses *snitch Snitch* (n.d.) in each of its nodes, which informs about the network topology, workload, historical latency conditions, and the detection of failing or slow nodes. Snitch allows Cassandra to distribute replicas according to the replication strategy by grouping machines into datacenters and racks. In this work, we use Cassandra as an example application (specifically, in our evaluations) to demonstrate that it has diverse requirements and that it is unaware of resource heterogeneity.

Programmable Switches and Network Telemetry. Monitoring the state of the network—also known as network telemetry—has been well-studied by the community for decades. Telemetry information is collected at different granularities (e.g., packets, flows, samples of flows, etc. *In-band Network Telemetry (INT)* (n.d.); *Linux SNMP Counter* (n.d.)) by installing network taps at key locations—along with switches—in the network. Recently, programmable switches Bosshart et al. (2014b) are slowly replacing the traditional setup; this is primarily due to their increased flexibility and functionality. The collected telemetry is sent either out-of-band (directly) or in-band (via dataplane packets to a "telemetry sink") to a remote inference engine or a controller and further actions are taken *In-band Network Telemetry (INT)* (n.d.); Li, Miao, Kim, and Yu (2016); Yu, Jose, and Miao (2013). Unlike traditional network telemetry, programmable switches enable collection and dissemination of processed telemetry reports that are succinct (e.g., reporting heavy hitters vs. sending raw packets or counters to the collector) and that captures the state of the network effectively.

Our research is motivated by the need to close the semantic gap between (a) the growing requirements of geo-distributed applications and the heterogeneity of the DC resources on which they are deployed as well as (b) the paucity of monitoring frameworks to capture fine-grained telemetry information at multiple dimensions (i.e., at the heterogeneous resource level and at the end-to-end application level).

Growing Resource Heterogeneity and Application Requirements. While applications are scaling across DCs that are geographically distributed, the hardware resources of DC systems are also becoming more and more

heterogeneous Legtchenko et al. (2017); Mars and Tang (2013). For example, take the case of storage heterogeneity: though DCs are moving towards an era of fast SSDs and nonvolatile memory (NVMe), traditional low cost-but-slower alternatives such as HDDs continue to be a vital part of a storage tier, thereby increasing storage heterogeneity across datacenters Mars and Tang (2013).

In addition, the heterogeneity of resources impacts application management, request routing, data placement, replica selection, and has a direct impact on applications' performance and requirements. For example, Cassandra is highly network-intensive as well as storage-intensive. Unfortunately, such an application must quickly decide on which replica to route to and what request to perform during data placement or fetching. Unfortunately, today's geo-distributed applications (1) are unaware of resource heterogeneity (e.g. storage SSD vs. hard disk), (2) network dynamism (e.g., routing delays, link outages, and path failures) Popescu, Zilberman, and Moore (2017), and (3) fine-grained host-level hardware metrics (e.g., the storage hardware's program-erase (P/E) cycles, temperature, I/O traffic), or software bottlenecks (e.g., application page cache state, storage I/O queue, TCP queue occupancy, segmentation Qdisc queue).These fine-grained metrics will be referred to as micrometrics.

Lack of Multi-dimensional Monitoring. The problem is complicated further by the lack of multi-dimensional resource monitoring frameworks. For example, deploying Cassandra in a WAN is fraught with challenges including WAN heterogeneity, path failures, unplanned outages, and performance and topological changes. Prior efforts attempt to address these challenges—*in isolation*—by creating topology awareness Jain et al. (2013), latency and bandwidth awareness Jonathan, Chandra, and Weissman (2018), network-level multi-dimensional resource monitoring and aggregation Dolberg, Francois, and Engel (2012), and snitching mechanisms *Snitch* (n.d.). While the above-mentioned efforts are as compelling as ever, they are mostly one-dimensional. Other multi-dimensional work such as Grandl, Ananthanarayanan, Kandula, Rao, and Akella (2014) deal with CPU and bandwidth heterogeneity for long-running stream processing systems and is not designed in the context of geo-distributed applications in general. In addition, these approaches are an ill-fit for latency-sensitive applications. Moreover, these approaches are mostly focused on application-level resource adaptability and a lack of high-resolution resource monitoring does not have a significant impact on performance.

On the network front, while the programmable switches and network telemetry efforts provide the needed micrometrics (e.g., path that a packet takes, number of unique flows per second, heavy hitters), they are network specific, and we require concerted effort between the programmable switches and host OSes.

CHAPTER III
MEASUREMENT TECHNIQUES

In this section, we explain two measurement techniques for active and passive network measurement. We begin by describing the in-depth implementation of the netflow to packet synthesizer, which we've termed `FlowSynth`, regarding how values are stored and sent across physical links and explain the respective results. We follow this section with the description of `eBPFlow`, a passive network flow collector based off of the Extended Berkeley Packet Filter, and the evaluation of its abilities.

FlowSynth

In this section, we describe the design and evaluation of FlowSynth, a kernel-based program we designed as a telemetry system for future network measurement tools.

Implementation. We were fortunate enough to acquire a dataset of 280 million recorded flows at the University of Oregon to use for our own testing purposes. Among these reported flows they contained the following categories delimited by commas:

1. Start Timestamp in Epoch Format
2. End Timestamp in Epoch Format
3. Source IP Address
4. End IP Address
5. Source Port Number
6. Destination Port Number (or a float type.code if ICMP/IGMP)
7. IP Protocol Number
8. Type of Service
9. Transmission Control Protocol Flags (default 0 if not TCP)
10. Number of Packets
11. Number of Bytes
12. Router Ingress Port
13. Router Egress Port
14. Source ASN

15. Destination ASN

Upon dissection of this data, we found that the top four protocols associated with the IP protocol field were ICMP, IGMP, TCP, and UDP. Since including all 255 protocols would be a tedious and not so prosperous task for our research, we decided to implement only these four protocols for our future flow to packet synthesizer.

We now approach the phase where our prior readings and programming methods take effect. we began by constructing simple packets with layer 1 being an Ethernet frame, layer 2 an IP header, layer 3 the UDP header, and a gibberish payload for layer 7. The netinet repository *Netinet Repository* (n.d.) from the FreeBSD C library provided me with all the headers we required, equipped with compact structures used as layer headers. Sockets *GNU C Library* (n.d.), provided by the GNU C Library, allowed me to reliably transfer packets onto the transmission queue of our NIC for packet sending. After tinkering with our program for some time, we found that data could be reliably transferred over the localhost interface using our prototype.

Unfortunately, the localhost interface is a very simplistic feedback loop interface which is an unreliable means of testing a program's efficiency and viability. Therefore, we decided to transfer our tests to CloudLabs Duplyakin et al. (2019), a cloud provisioning platform that grants you access to a multitude of nodes for network testing. After pulling our repository onto the client node and issuing a TCPdump on a linked interface using our server node, we found that if the source and destination MAC addresses were specified alongside a network interface, our software could reliably transfer packets from client to server.

With our prototype finished, it wasn't too difficult to implement the rest of the protocols since all we had to do was replace the UDP header with a desired header, and configure it according to the flow entry in the dataset. Once we had implemented ICMP, IGMP, and TCP, we were ready to move onto time related issues.

The question became: How could we read flows from a dataset and send packets that replicated the flow over a set duration of time? We began by implementing this concept of having our program's run-time correlate with the time from the dataset. We did this by creating a method which reads the first entry from the dataset, grabs the start time of the packet, and subtracts it from every subsequent flow in the dataset. This way, all flow times have been neutralized to program time, which makes it much simpler to decide when to read a new flow; My program should only read a new flow if the start time is before or equal to our program's current time.

Now that our program knows when to read flows, it must know when to send from the flow as well. We achieved this goal by taking the net time the flow was recorded for and dividing that value by the net amount of packets the flow had, *(endTime - startTime) / packets*; This was stored in a variable. We decided to

create a specific structure that would encompass details about the flow for low overhead sending. Therefore, we decided that when each flow is read, it would then be stored as custom packet structure named *grand_packet* with the following attributes:

1. *char *buffer - Used* for storing the contents of the actual packet itself
2. *unsigned int packets_left* - Tracks how many packets are left in the entry
3. *float d_time* - Holds the calculated delta time between packet transmission, calculated using the method shown above
4. *double cur_time* - Tracks the current time state of the packet for scheduling
5. *length* - Holds the total length of the packet for socket transmission
6. *struct grand_packet *last* - Used for network scheduling and points to previous packet
7. *struct grand_packet *next* - Used for network scheduling and points to next packet

Now, we can use the program's time in contrast with the flow's *cur_time* to determine if another packet should be sent. If the flow should be sent at some program run-time, then a packet is sent using the flow's buffer and add the flow's *d_time* to its *cur_time*, 1 is subtracted from *packets_left*, and then program continues to monitor the flow. Once the flow's *packets_left* reaches 0, the memory addresses that held the flow's structure are freed.

Lastly, we must now consider that the dataset will have many flows whose start time will fall within the program's current time – We found a max of half a million flows that fell victim from this particular dataset! Therefore, we needed some sort of scheduler that would check each applicable flow and see if it was time for that flow to send, do nothing, or free its own memory. We resolved this issue by applying a Round-Robin (RR) algorithm to network scheduling, where each flow would be designated a small amount of time, where the RR would then continue onto the next flow in the circle. Therefore, when each new flow is implemented to the RR circle, we set the first flow's **last* to point to the new flow, and the last flow's **next* to point to the new flow as well. This way, we have a linked list of flows that the RR can easily traverse and send from when required.

Evaluation. My flow to packet generator software never fell behind when generating packets for the University of Oregon dataset we supplied it with. Therefore, we decided to test the limits of our program to get a range of benchmarks for potential usages.

We began by creating test flows for our software using a lightweight flow generating script. This script was utilized by automated testing, a bash shell

Figure 2. Netflow to Packet Synthesizer's performance in PPS, run on Intel Xeon D-1548 2.0 GHz core, MLNX X-3 10 Gb/s NIC.

script we created which creates flows with byte rates up to link-rate (generally 10 Gb/s for most NICs). The performance of our flow to packet generator, called UONetflowC, are shown below.

Here, we can see that our program tends to decline in accuracy around 900 kpps. Granted the number of packet options our program accommodates for, how it runs entirely in kernel space, and that we use a single core to transmit all packets, these results are rather impressive. In fact, our results 2 compete fairly well with some of the leading kernel space packet synthesizers 3, such as TCPReplay *Pcap Editing and Replaying Utilities* (n.d.).

We witness that TCPReplay's max PPS is around 950 kpps, and on a single core, 825 kpps. Therefore, our program takes the win on a single core, however without multi-threading capabilities, it falls short in a multiple core comparison.

We ran another test to see what sort of bit rates our flow to packet generator could achieve. All packets must abide by an MTU of 1500, therefore a realistic max packet size that wouldn't be dropped by the kernel is approximately 1300 bits. With this method, we was able to achieve line-rate BPS on our node's 10 Gb/s NIC.

Summary. As the main purpose of this synthesizer was to act as a telemetry system for future Self-DN capability research and production, it satisfies this requirement by being able to achieve link-rate transmission. Although outperformed by kernel bypass packet generators such as *pktgen* and *DPDK Data Plane Development Kit* (n.d.); Turull, Sjödin, and Olsson (2016), the simplistic flow creation mechanism entailed by our repository allows an accurate replication of very specific user-defined flows, enabling intricate testing of network related

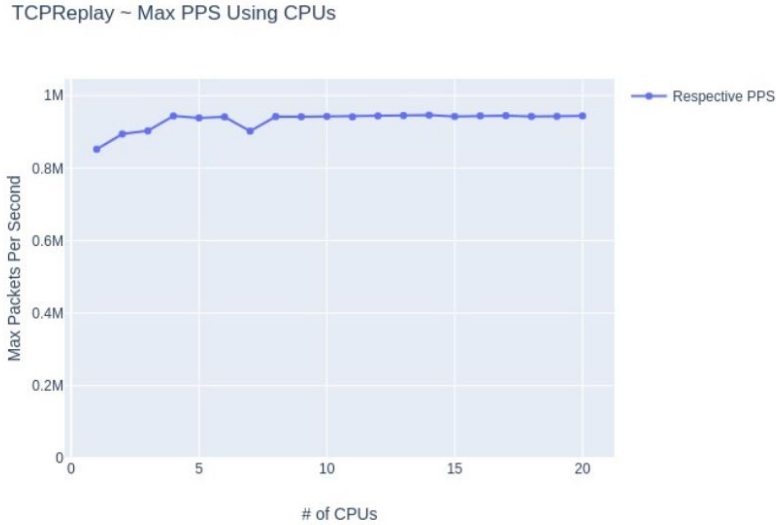

Figure 3. TCPReplay to Packet Synthesizer's performance in PPS, run on Intel Xeon D-1548 2.0 GHz core, MLNX X-3 10 Gb/s NIC.

software. In fact, this project was heavily utilized for the first half of `eBPFlow` until stress testing was required.

eBPFlow

In this section, we describe the design and evaluation of eBPFlow, an XDP-based program we designed to test the efficiency and accuracy of passive eBPF flow collection.

Benefits. We motivate the design of eBPFlow by outlining the benefits of XDP-based programs over state-of-the-art software and hardware flow collectors.

First, XDP provides an in-kernel system for packet processing by providing access to the raw packet memory space, making it optimal for flow recording. If flow recording is performed along with packet forwarding (i.e., packets are not just dropped after being recorded), competing frameworks such as DPDK incur overhead as a result of bypassing kernel space due to the required re-injection of packets into kernel space for sending. XDP, however, does not change the native path of packet processing but instead acts as an intermediate recording mechanism that resides between the NIC and kernel space. Furthermore, since XDP provides access to the raw packet memory space, the recording is done before any memory allocation required by the network stack, obviating the need for packet parsing or a socket buffer.

Second, XDP is integrated cooperatively with the regular networking stack, making it much more adaptable and compatible with a wide variety of hardware. Other services such as DPDK and Netmap limit their potential users due to their strict hardware dependencies *Data Plane Development Kit* (n.d.); Rizzo (2012),

although incompatible Netmap NICs can still use emulation at a cost of decreased performance. This is key in enhancing the generality of eBPFlow.

Third, XDP can be dynamically reprogrammed without any interruption to packet processing, allowing features and policies to be easily added or removed. This dynamic nature of swapping driver-space programs is unprecedented in current flow collecting services and makes the flow collection via eBPFlow more responsive to changing needs. On the contrary, hardware- and router-based flow collectors are rigid and can be cost-prohibitive.

Finally, XDP incurs lower CPU usage compared to other software-oriented flow collecting services due to the minimized number of cores allocated to packet processing. This hinges on the fact that the XDP program resides in driver space and records packet attributes before any memory allocation required by the networking stack. Lower CPU usage equates to better overall computational performance and power-saving.

Implementation. Motivated by these benefits, we implement eBPFlow—an XDP-based program spanning across user and kernel space.

User Space. The user space of eBPFlow is a Python script that imports the BPF Compiler Collection (BCC) library, exposing BPF Python objects that compatibly operate with the native BCC library. The script loads and compiles our program (i.e., eBPFlow), installs it in XDP, and establishes maps that allow information to be shared between the user and driver space modules. In particular, per-CPU maps are created to store flows, allowing multiple cores to operate atomically on identically hashed flow records. The keys used to the BPF maps to store flows are a custom custom structure that includes static flow-level attributes such as source/destination IPs and ports. Naturally, the value corresponding to a given flow key holds the number of bytes and packets received, timestamps, etc. A user-level argument provided to this script is the aggregation time; once a flow has sat idle (i.e., received no further packets) for a duration that exceeds the aggregation time, the flow becomes cached by moving to another per-CPU hashmap and deleting the entry in the actively used flow accumulation hashmap. In this sense, the Python script acts as a garbage collector for expired flows to optimize storage and lookup efficiency. The userspace Python program, upon copying flows from kernel BPF maps to user space, merges the per-CPU records to form a single coherent accounting of a given flow.

Kernel Space. Complementary to the user space script, eBPFlow, written in C and using eBPF helper functions, is mounted on the XDP kernel hook. Like other XDP programs, all non-mapped memory accessed or manipulated must be found within the space of an incoming packet. In the XDP code, we parse the various layers of the packet, extract and construct a key for the BPF map, and use that to update a flow record. Care is taken to minimize the number of map operations in order to scale well with high packet rates

Challenges. In the design and implementation of eBPFlow, we tackled two key challenges: (a) the limitations of implementing code within the driver space, and (b) storage limitations with BPF maps. While the first limitation comes with the territory of coding within the kernel and within the constraints of the BPF safety checker, it is worth mentioning. For example, and as noted earlier, no loops may be used within the program, although *#pragma unroll* can unroll definite loops within the BPF C code as a workaround. With eBPFlow, this issue arises particularly in unrolling VLAN headers. The second limitation regarding maximum size of BPF maps is imposes an artificial ceiling on the number of flows that can be captured within a single map, as we discuss later in this section.

Evaluation. In this section we describe our experiments to evaluate the performance of eBPFlow.

Configuration. We design our experiment in the CloudLab infrastructure Duplyakin et al. (2019), where two nodes are connected by an Ethernet link. Both nodes reside within the Utah cluster and have 2.4 GHz 10-core Intel E5-2640v4 processors with 64 GB ECC memory, and two Dual-port Mellanox ConnectX4 25 Gb/s NICs. We denote one node as the flow collector, running eBPFlow. The other node uses pktgen Turull et al. (2016), a high-speed packet generation tool, to emit packets to stress the flow collector. We created a pktgen script to cause each core to emit 60 million 60 byte UDP packets at a rate of 1 Mpps (1 Million packets per second). Moreover, this script also randomly generated destination IPv4 addresses from a /24 prefix, as well as randomizing the source port. The randomization of addresses and ports results in 25K flows/sec emitted from pktgen. Lastly, we utilized a range of number of cores on the host running pktgen to provide offered loads from 1 Mpps up to 20 Mpps (just over 10 Gb/s, at maximum).

Results. We evaluated the performance of eBPFlow on two criteria: performance and accuracy. In particular, our main metric was the maximum offered packet rate that could be sustained with zero or negligible packet loss (i.e., less than 0.0001% loss) at the flow collector. By simply comparing the number of packets emitted from pktgen with the number of packets received by eBPFlow, we could assess whether a given packet rate could be sustained. In our experiments, we varied the number of CPU cores available on the host running eBPFlow as we also vary the offered packet rate from the pktgen host.

Figure 9 shows results of the number of CPU cores required (x axis) to sustain a packet rate (y axis, in Mpps). We observe in the plot that a single core can handle approximately 3.66 Mpps, and that this rate scales well with additional cores. The maximum rate we considered in our experiments, 20 Mpps, was achieved with 5 cores. Note that the number of flows/sec generated remains constant for these experiments, at 25K flows/sec.

In additional experiments, we evaluated the effect of increasing the number of flows generated from pktgen beyond the baseline 25K flows/sec that we used

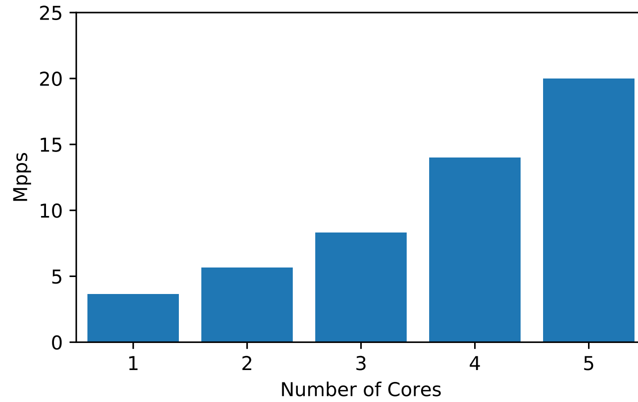

Figure 4. Maximum sustainable packet rate (Mpps) by eBPFlow for an increasing number of CPU cores.

in the experiments for Figure 9. Specifically, we increased the size of the prefix used for generating random destination addresses while holding the offered packet rate at 20 Mpps with 60B packets. For 5 cores, eBPFlow could sustain 100K flows/sec, with 6 cores 200K flows/sec, with 8 cores 400K flows/sec, and with 10 cores 800K flows/sec. At the 800K flows/sec mark, we encountered a limitation with the maximum size of eBPF maps.

CHAPTER IV
MONITORING FRAMEWORK

Implementation of MicroMon. In what follows, we describe the design of MicroMon. We first describe the basic set of micro-metrics, then briefly discuss the techniques to capture such micro-metrics, followed by ways to reduce collection and dissemination overheads. We then discuss our proposal for scalable inference.

Micrometrics Selection. A key technical challenge towards the design of MicroMon lies in identifying and selecting the key software- and hardware-based micrometrics to collect from all the resources (e.g., network, storage).

Mapping Micrometrics to Resource Sensitivity. One problem is that the choice of micrometrics could substantially vary across applications, applications perceived performance-metrics, and DC/cloud deployments. For example, latency sensitive microservices, and geo-distributed key-value stores like Cassandra and memcached Ousterhout, Fried, Behrens, Belay, and Balakrishnan (2019) are highly latency sensitive and are directly dependent on I/O latency with short request times compared to previously studied geo-analytics such as Apache Spark that use compute-intensive and throughput sensitive queues. we observe that one way to scope the problem of micrometrics selection is by mapping applications to hardware and software resource sensitivity.

Storage H/W and S/W Micrometrics. To scope the solution in our current design of MicroMon, we focus on I/O-intensive Cassandra in this paper. Without loss of generality, these metrics are applicable to several large classes of I/O-intensive applications.

Interplay between hardware micrometrics. Most DCs monitor and collect system (and storage) health and performance metrics. For storage, this includes straightforward metrics such as latency and bandwidth as well as device-level SMART counters such as Raw Read Error Rate (read error rate), Program Fail Count (write error count), device block wear, and temperature. Note that these metrics are used by prior studies in isolation towards data placement and load imbalance Narayanan et al. (2016a). We posit the importance of considering the interplay between such hardware micrometrics, network performance, and application's data access patterns. For example, in NoSQL store replica selection and request routing, a node with low network latency but high program error (PE) count can still be used as a reliable read replica as long as read error rates and temperature do not increase. Unfortunately, *current one-dimensional monitoring systems simply quarantine the entire physical node with high PE, routing requests to nodes with higher network latency* Narayanan et al. (2016a).

Interplay between Software micrometrics. Storage software micrometrics such as per-node page cache and outstanding block I/O requests could play a crucial role towards request routing and replica selection. For example, in

Cassandra's LSM design, the software storage is maintained as String Sorted Tables (SST tables) composed of several files, with each file storing a range of key-value pairs. As we will discuss in our evaluation, accessing data from a replica with hard-drives but significantly large page cache state can significantly boost throughput and lower latency compared to accessing data from a SSD replica without pagecache. However, *current monitoring and replica selection mechanisms clearly lack such semantic awareness.*

Network H/W and S/W micrometrics. While several metrics are available via Simple Network Management Protocol (SNMP) including sensor information (e.g., temperature, fan status, etc.), collections of hardware and software micrometrics from the network *and* joint decision with host-specific micrometrics are critical to the success of our work. To this end, we identify several network hardware-specific micrometrics including status of relevant links, SFP status (if available); resource information (e.g., CPU load, memory usage) via SNMP get, up/down status of devices/port, queue and buffer utilization, and link-specific information including Q-drops, SNR; among others. For example, prior effort utilized the signal quality information (i.e., Q-drop) to predict future outages in large optical backbones Ghobadi and Mahajan (2016). Indeed, we propose to leverage these micrometrics and expose them to applications.

Similar to the hardware micrometrics, recent innovations in programmable data planes support seamless extraction of software micrometrics from the network including port violation, QoS statistics including drops per queue, packets per Differentiated Services Code Point (DSCP), packet errors and discards, and aggregated network states including heavy hitters, flow arrival rate, etc. to further enhance the performance of applications.

MicroMon Components We develop MicroMon, a replacement of Cassandra's Snitch mechanism for fine-grained micrometrics collection and multi-dimensional resource monitoring as shown in Figure 5. As discussed earlier in § II, in Cassandra, each node gets assigned to some key range and act as a coordinator as well as replica for other set of keys. Hence, in each node, a user-level component—*MicroMon-engine*—integrated with Cassandra, is responsible for collecting micrometrics from its own node as well as replicas. In addition to the user-level component, MicroMon also contains an OS-level driver (*MicroMon-driver*) to monitor hardware micrometrics (e.g., SMART counters Narayanan et al. (2016b) such as high P/E warnings, NIC packet drops) and software micrometrics (e.g., application's page cache state, storage and network I/O queue delays, TCP queue occupancy). Further, our choice of extending snitch as opposed to designing a new tool is mainly because of Snitch's wide usage and to avoid polluting replicas with new monitoring tools.

Micrometrics Collection & Dissemination. We next focus on designing techniques to reduce micrometrics collection and dissemination

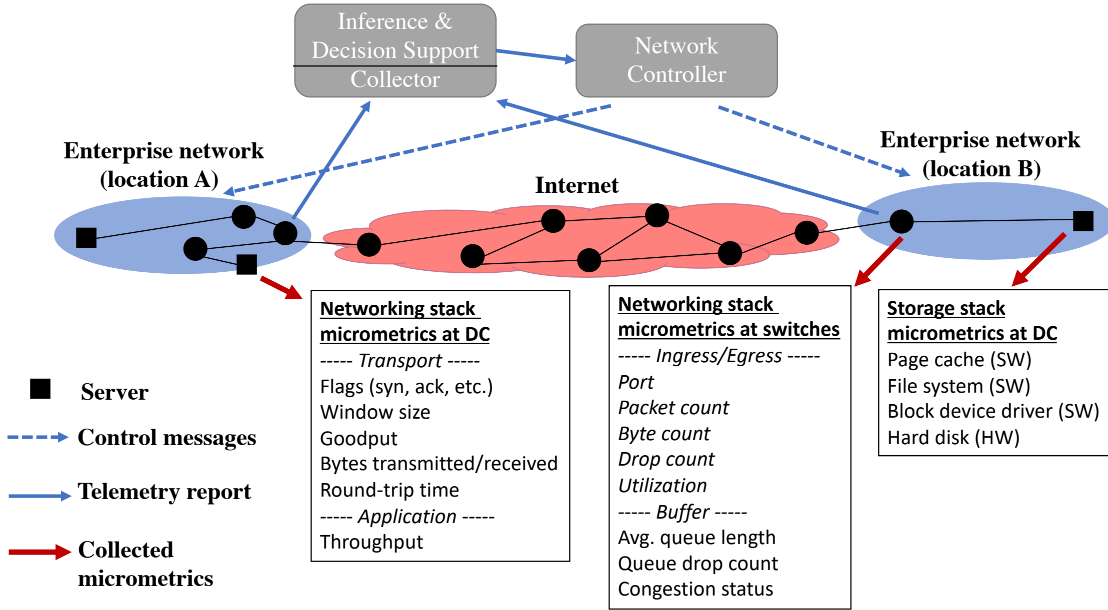

Figure 5. **High-level MicroMon Design.** Figure shows the integration of our HW and SW micro-metrics collection and dissemination in MicroMon.

overheads, since collection and dissemination happens at both hosts and the network.

Host-level collection with Anomaly Reports. One issue complicates the collection of micrometrics at the host level. Specifically, the number of micrometrics increases with the number of hardware resources and software subsystems used by an application. For example, even a single SSD's SMART counters contain close to 32 counters that can impact performance Narayanan et al. (2016b). This, coupled with software micrometrics, could significantly impact the resolution at which this information is collected and disseminated, micrometrics in data plane at high resolution could impact impacting the performance of applications. To address the issue, we propose anomaly reports, where for all possible host-level micrometrics, the host OS only reports anomalies. Such reports require no further processing at the inference and decision engine (explained below). We argue that host OSes can already identify software and hardware anomalies, given a better holistic view of the system. For example, monitoring anomalies such as high page cache misses, network queue latency or high SSD P/E count can be done at the host, only reporting anomalies rather than processing them. Further, we propose to extend the *MicroMon-driver* to report such anomalies. *MicroMon-driver* periodically collects the required software and hardware micrometrics and composes a monitoring packet with anomalies.

Co-designing Network-level Dissemination with Host OSes. We leverage the innovations in programmable switches and co-design two mechanisms (with host OSes) to extend network telemetry to support heterogeneous resource monitoring. our first mechanism (a) generates monitoring packets using MicroMon-driver

with anomalies identified at hosts as payload and (b) uses in-band network telemetry (INT), subsequently, to add network micrometrics (identified in § **??**) to those monitoring packets and disseminate them via sink nodes to the decision engine. In a general INT model, data packets contain headers which in turn contain instructions for the traffic sources (e.g., applications, end-host networking stacks) about what state to collect and write into the packets as it transits the network. However, given the limitations of INT packet metadata sizes and INT instruction fields (16-bits) , and the possibility of hundreds of micrometrics, we leverage anomaly reports and using techniques such as run-length encoding and memoization for the ones collected at the network. Telemetry reports can be generated at switches based on pre-established anomalies, opening up the possibility of reporting only aggregated network events and naturally overcoming the micrometrics collection and dissemination. For geo-distributed deployments as shown in Figure 5, we plan to leverage INT over any encapsulation (e.g., over TCP/UDP, depending on the application) to collect and disseminate micrometrics.

our second mechanism is very similar to the first one, except the usage of INT in the dissemination step above. Specifically, programmable switches also support out-of-band reporting of telemetry reports directly to the decision engine. This is to overcome the difficulties in strategic placement of INT sink nodes in an enterprise WAN. In light of this, we support out-of-band telemetry reports containing aggregated network and host micrometrics to be reported directly to decision engine.

Scalable Inference Logic. Next, we discuss our preliminary scoring-based inference in Cassandra and then discuss our ongoing work on building a generalized inference logic.

Scoring-based Inference. The current snitching mechanism in Cassandra uses a scoring mechanism that sorts the endpoints by their latency with an adaptive replica failure detector. The coordinator node initially sorts the replicas based on their network proximity and then uses response latency to update the score frequently. Instead, in MicroMon, We modify the scoring mechanism to consider different software and hardware storage and network micro-metrics in addition to straight forward network latency and bandwidth. My current prototypic scoring mechanism (as evaluated), assigns equal weights to all software and hardware micro-metrics and higher weights to network latency and bandwidth.

Experimental Evaluation. To understand the performance benefits and implication of our proposed MicroMon, we next compare the performance of MicroMon with the vanilla dynamic snitching mechanism in Cassandra, under different deployments, by varying multidimensional micrometrics such as storage heterogeneity (SSD vs. HDD), application threads, network latencies across replicas, and the page cache.

Benchmark. We run the industry standard and widely-used YCSB Cooper, Silberstein, Tam, Ramakrishnan, and Sears (2010) benchmark with Cassandra.

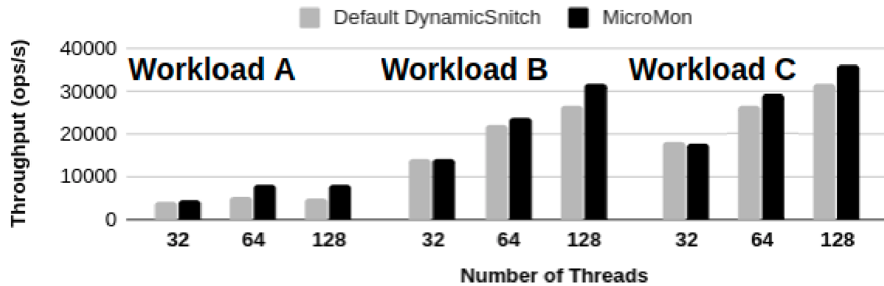

Figure 6. **Storage Hardware Impact.** Results for YCSB workloads varying the number of threads and the storage hardware across replicas.

Due to space restrictions, we study three key-value access patterns from the YCSB cloud suite: (1) Workload A (a write-heavy workload with 50/50 read-write ratio), Workload B with 95% read, and Workload C (a read-only workload). We show the run phase of YCSB for 95th percentile latency and throughput for 500K operations with a runtime of around 30 minutes.

Experimental Setup. We design our experiments in the CloudLab infrastructure with three physical nodes located within the Utah and Utah-APT clusters. For the SSD replica located in the Utah cluster, we use a system with a 2.0GHz, 8-core Intel Xeon D-1548 processor and 64 GB ECC memory. For the other two replicas residing in the Utah-APT cluster, their systems entail a 2.1GHz, 8-core Intel Xeon E5-2450 processor with 16 GB RDIMM memory. The keys are mapped across these three Cassandra nodes, where each node acts as a coordinator for a range of keys with other nodes acting as a replica. Two out of three Cassandra nodes use a hard-disk drive with 1-2 ms random-access latency and 5-15 MB/s random access bandwidth compared to the SSD node with 10-30 us latency and 300 MB/s bandwidth. The average network latency between nodes solely within Utah-APT is around 0.25-0.35 ms latency, and between nodes from Utah-APT and Utah is around 0.35-0.45ms latency. In all our experiments, We compare MicroMon against the default dynamic snitching mechanism that mainly considers network latency.

Impact of Heterogeneous Storage. First, to understand the impact of considering underlying storage hardware's latency and bandwidth, we compare the default dynamic-snitching mechanism in Cassandra against MicroMon. For avoiding undue overheads of network latency, we maintain two replicas of Cassandra at the same datacenter, with one replica using SSD and other replica using hard-disk to serve YCSB requests. We maintain Cassandra's in-memory buffer (skiplist) size to the default 64MB and commit the log (using *fsync*) every 50ms (a parameter in Cassandra).

Figure 6 shows the throughput of YCSB workloads A, B, and C in the y-axis, and the number of client threads issuing requests on the x-axis. The client issue 1KB requests totaling 500K operations. First, the dynamic snitch in Cassandra

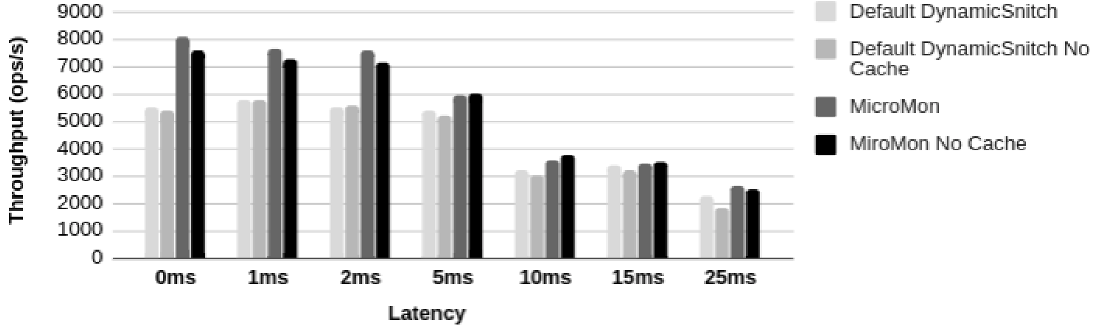

Figure 7. **Network and Storage Latency Impact.** Results show the combined considering network and storage latency for YCSB workload A with 16-threads.

is oblivious to storage latencies and only considers network latencies. Due to the lack of network latencies, it sends even requests to both the SSD and HDD replicas. In contrast, MicroMon also considers storage latency through the micro-metric collection, redirecting a significant number of requests to SSD replica. Workload A with 50% read and writes shows higher gains over workload B with 5% writes. Further, with increasing client threads (in the x-axis), the MicroMon gains improve by exploiting SSDs higher paallelism, leading to 49% gains for workload A. Finally, for read-only workload C combined with the YCSB's uniform distribution, the benefits are lower for our current evaluation scale. Also, note that we only report the run-phase and not the warmup phase (cold access). *The results highlight the need for multi-dimension micrometrics, which includes storage hardware monitoring.*

Network Latency and Page Cache. To understand the combined impact of storage and network heterogeneity, in Figure 7, we compare the Default DynamicSnitch with our MicroMon based on synthesized latency toward the SSD replica, whereas the HDD replica's network latency ($< 1ms$) is unchanged. In the x-axis, we gradually increase the network latency of SSD from $0ms$ and $25ms$. We include two types of comparisons with respect to the SSD node, one where the cache remains as default and the other where the SSD replica has a lower page cache footprint (MicroMon No Cache and Default DynamicSnitch No Cache) simulated by frequently clearing the cache. For brevity, we only show YCSB's workload A (50% write and read) for 64 threads.

First, without any added network latency, the SSD replica in a remote cluster provides a significant throughput (40% increase; see "Default DynamicSnitch" and "MicroMon" bars in Figure 7) compared to the Utah-APT replicas. However, when adding $5ms$ latency to the SSD replica's network latency, the throughput reduces when compared to the local HDD replica, but still maintaining a considerable (10%) throughput increase. Next, regarding the page cache, for the

SSD replica with lower page-cache (see "No Cache" bars), we notice that even without network latency, MicroMon prioritizes the Utah-APT nodes a little more than the default DynamicSnitch resulting in slightly higher average throughput (37% increase; see "Default DynamicSnitch No Cache" and "MicroMon No Cache" bars in Figure 7). In case of increase in network latencies, we observe a marginal decrease in throughput gains. For example, while we observe a throughput gain of 33% using MicroMon for $2ms$ network latency, the throughput gain decreases to 24% in case of $25ms$ of latency. These results highlight the need for multidimensional monitoring.

CHAPTER V
CONCLUSION

Summary of Results

FlowSynth. For this unique telemetry system, we successfully created a flow to packet generator capable of reading from flow datasets and generating realistic packets accordingly. The generator uses a system of adjusting recorded flow time frames to that of the program's time, and sending in a fashion that perfectly replicates each flow. The flow to packet generator was able to reach a single core, kernel space packets per second of 900 kpps, and link-rate BPS through 1300 byte sized packets. This data shows that our software competes well against state of the art kernel space packet generators such as *TCPReplay*. Upon testing this software as a telemetry system for `eBPFlow`, I've been able to verify its helpfulness in evaluating both the efficiency and accuracy of new capabilities for Self-DNs.

eBPFlow. In this project we investigate the Extended Berkeley Packet Filter (eBPF) as an implementation path for active and passive network measurement. We describe the eBPF architecture, programming model, and APIs in the network measurement context, focusing specifically on the BPF Compiler Collection (bcc) as a more user-friendly API. We discuss how eBPF/bcc APIs can be used for both passive and active measurement, and describe how current limitations and safety requirements impose certain constraints on measurement algorithms currently possible. We illustrate the potential for high-fidelity active and passive network measurement with eBPF by developing a passive flow collector called eBPFlow and evaluate it in a controlled setting. eBPFlow clearly scales well to perform full flow capture at 10 Gb/s on 5 cores with 60 byte packets, and with no special hardware acceleration or storage configuration. Upon further evaluation, we find that `eBPFlow` is able to record approximately 800K network flows per second, confined only by the space restriction of maximum sized eBPF maps.

MicroMon. In this project we present MicroMon, a multi-dimensional monitoring and inference framework for geo-distributed applications using heterogeneous hardware. Using micrometrics, a set of fine-grained hardware and software metrics, we are able to study the combined impact of heterogeneous WAN and storage resources on Cassandra's performance. To reduce micrometrics collection and dissemination overheads, we propose anomaly reports and concerted effort between the programmable switches and host OSes to reduce the overhead of collecting and disseminating thousands of micrometrics in WAN. My prototype deployed in a geo-distributed Cassandra using heterogeneous storage and network shows close to 49% performance gains.

Lessons Learned

FlowSynth. Although out-shined by packet generators such as Netmap and DPDK, FlowSynth provides a unique packet synthesizing process by allowing the user to define the specifications of their desired flows. This has allowed us to test a multitude of edge cases entailed by `eBPFlow` and `MicroMon` by synthesizing flows specific to a packet that evaluates edge case conditions. We learn that a kernel-space packet generator, though versatile in that its compatibility with systems ranges far and wide, is not the most optimal architecture for stress testing telemetry systems.

eBPFlow. Overall, our experiences and the results of our experiments described in this paper strongly suggest eBPF as a promising high-fidelity vehicle for implementing passive and active measurement methods. There are, however, some limitations that we have encountered. In particular, one needs to take care in regards to the movement of data from kernel to userspace. There are facilities for *pushing* data from kernel to userspace (perf buffers *bcc Reference Guide* (n.d.)) which cause callbacks in the control program (*e.g.*, in Python), but data can also be *pulled* through direct access to eBPF maps from a control program. While the perf interface is simple, it relies on a fixed-size ring buffer that can be overwhelmed if the data rate is too high. Direct map access does not suffer from that issue but requires careful data management. Future APIs may allow event batching through the perf interface, but that is not currently possible. Another challenge has to do with debugging, which largely relies on printf-style tracing. Silent failures can happen (particularly, in our experience, with XDP), which can be difficult to identify, and memory safety checks can sometimes be difficult to understand and fix. We do, however, expect debugging tools to improve as the eBPF subsystem matures.

We find that passive measurement is the more obvious candidate for eBPF, and besides packet and flow capture, it is possible to trace arbitrary kernel functions related to the networking stack and arbitrary functions in user mode programs, in addition to tracing events from well-defined tracepoints Gregg (2019). For example, there are numerous short programs and even “one-liners” to track TCP connection state, retransmits, socket activity, etc. *BCC tools* (n.d.); *bpftrace* (n.d.); *A curated list of awesome projects related to eBPF* (n.d.); Gregg (2019); *XDP-Project* (n.d.). While these tasks are already achievable with other tools, eBPF offers a standard interface for doing such tracing efficiently, safely, and with broad visibility.

MicroMon. Micrometric monitoring is a highly important concept that has received very little attention since the introduction of heterogeneous networks. Through our results, we have found that using micrometrics to influence which nodes of a linked network should receive and deploy requests yields a considerable throughput increase. Although our experiments are solely focused on Apache Cassandra within this paper, our research and findings are applicable to the vast

majority of third party services which utilize load balancing on a multitude of heterogeneous systems.

Future Work

FlowSynth. At the cost of versatility, we plan on improving the throughput of FlowSynth by migrating its services to the framework of DPDK. As exemplified by the 60+ Mpps achievable by MoonGen Emmerich (2014), a packet generator based on the DPDK framework, we have high hopes for the future status of `FlowSynth` as a future intermediate telemetry tool.

eBPFlow. We believe that eBPF holds a lot of promise for both active and passive network measurement, and there are several directions we intend to investigate in future work. In particular, we plan to revisit the efforts of Govindan and Paxson to examine delays in ICMP packet generation at routers Govindan and Paxson (2002) in an effort to better understand how to reduce or eliminate noise in the hop-limited latency measurements. We also plan to collect a much broader set of measurements to better understand wide-area queuing dynamics and congestion. Moreover, we plan to examine alternative data organization for storing flows beyond current eBPF-imposed limits. Finally, we plan to explore additional ways in which eBPF may be used for active and passive network measurement, including new ways to support measurement within eBPF.

MicroMon. Only capturing currently available hardware and software micrometrics (for network and storage) is insufficient. This calls for consideration of broader class of (multi-dimensional) resources (e.g., compute, memory, network, storage). Further, identifying and innovating new microarchitecture-level micrometrics that capture the impact of multi-dimensional resource usage holistically is critical. For example, no micrometrics exists to capture the hardware overheads in using the same PCI channels for a storage intensive and network-intensive application. We believe a cross-stack fine-grained heterogeneous monitoring of memory, storage, network, and compute requires microarchitecture-level innovations in the hardware and new software micrometrics. We plan to consider microarchitectural innovations for MicroMon as part of future work. In addition, the decision of what micrometrics to use must be automated without relying on administrators, application developers or network operators. Finally, the current prototype of MicroMon assigns equal weights to all software and hardware micrometrics. Unfortunately, this is not applicable for all the applications. For example, micrometrics for bandwidth-sensitive applications require higher weights for micrometrics collected from the WAN than those collected from the end hosts. Profiling the weights micrometrics for diverse applications atop heterogeneous resources is a complex problem. We intend to consider this in MicroMon as part of future work.

REFERENCES CITED

Ahmed, Z., Alizai, M. H., & Syed, A. A. (2018). Inkev: In-kernel distributed network virtualization for dcn. *ACM SIGCOMM Computer Communication Review*, *46*(3), 4.

Apache Cassandra. (n.d.). (http://cassandra.apache.org/)

bcc Reference Guide. (n.d.). https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md. (Accessed May 2020)

BCC tools. (n.d.). https://github.com/iovisor/bcc/tree/master/tools. (Accessed May 2020)

Begel, A., McCanne, S., & Graham, S. L. (1999). Bpf exploiting global data-flow optimization in a generalized packet filter architecture. In *Proceedings of the conference on applications, technologies, architectures, and protocols for computer communication* (pp. 123–134).

Bertin, G. (2017). XDP in practice: integrating XDP into our DDoS mitigation pipeline. In *Netdev conference.*

Bonaventure, O. (2019). Making our networking stack truly extensible. In *2019 ieee 44th lcn symposium on emerging topics in networking (lcn symposium)* (pp. i–i).

Borkmann, D. (2016). On getting tc classifier fully programmable with cls bpf.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., ... others (2014a). P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, *44*(3), 87–95.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., ... Walker, D. (2014b, July). P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, *44*(3), 87–95. Retrieved from https://doi.org/10.1145/2656877.2656890 doi: 10.1145/2656877.2656890

BPF and XDP Reference Guide. (n.d.). http://docs.cilium.io/en/latest/bpf/. (Accessed May 2020)

BPF Features by Linux Kernel Version. (n.d.). https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md. (Accessed May 2020)

The BPF system call API, version 14. (n.d.). https://lwn.net/Articles/612878/. (Accessed May 2020)

bpftrace. (n.d.). https://github.com/iovisor/bpftrace. (Accessed May 2020)

Braden, R. T. (1988). A pseudo-machine for packet monitoring and statistics. *ACM SIGCOMM Computer Communication Review*, *18*(4), 200–209.

Cardigliano, A., Deri, L., Gasparakis, J., & Fusco, F. (2011). vpf_ring: Towards wire-speed network monitoring using virtual machines. In *Proceedings of the 2011 acm sigcomm conference on internet measurement conference* (pp. 533–548).

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010). Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st acm symposium on cloud computing.* Indianapolis, Indiana, USA.

Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., ... Woodford, D. (2013, August). Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, *31*(3). Retrieved from https://doi.org/10.1145/2491245 doi: 10.1145/2491245

Crovella, M., & Krishnamurthy, B. (2006). *Internet measurement: infrastructure, traffic and applications.* John Wiley & Sons, Inc.

A curated list of awesome projects related to eBPF. (n.d.). https://github.com/zoidbergwill/awesome-ebpf. (Accessed May 2020)

Data Plane Development Kit. (n.d.). https://www.dpdk.org/. (Accessed May 2020)

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ... Vogels, W. (2007). Dynamo: amazon's highly available key-value store. In *Acm sigops operating systems review* (Vol. 41, pp. 205–220).

Dolberg, L., Francois, J., & Engel, T. (2012). Efficient multidimensional aggregation for large scale monitoring. In *Presented as part of the 26th large installation system administration conference (LISA 12)* (pp. 163–180). San Diego, CA: USENIX. Retrieved from https://www.usenix.org/conference/lisa12/technical-sessions/presentation/dolberg

Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., ... Mishra, P. (2019, July). The design and operation of CloudLab. In *Proceedings of the USENIX annual technical conference (atc)* (pp. 1–14). Retrieved from https://www.flux.utah.edu/paper/duplyakin-atc19

ebpf - extended berkeley packet filter. (n.d.). https://www.iovisor.org/technology/ebpf.

Emmerich, P. (2014). *Moongen.* https://github.com/emmericp/MoonGen.

Engler, D. R., Kaashoek, M. F., & O'Toole Jr, J. (1995). Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, *29*(5), 251–266.

Feamster, N., Rexford, J., & Zegura, E. (2013). The road to sdn. *Queue*, *11*(12), 20–40.

Fusco, F., & Deri, L. (2010). High speed network traffic analysis with commodity multi-core systems. In *Proceedings of the 10th acm sigcomm conference on internet measurement* (pp. 218–224).

Ganger, G. R., Engler, D. R., Kaashoek, M. F., Briceno, H. M., Hunt, R., & Pinckney, T. (2002). Fast and flexible application-level networking on exokernel systems. *ACM Transactions on Computer Systems (TOCS)*, *20*(1), 49–83.

Ghobadi, M., & Mahajan, R. (2016). Optical layer failures in a large backbone. In *Proceedings of the 2016 internet measurement conference* (pp. 461–467).

Gnu c library. (n.d.). https://github.com/torvalds/linux/blob/master/include/linux/socket.h.

Govindan, R., & Paxson, V. (2002). Estimating router ICMP generation delays. In *Passive & active measurement (pam).*

Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., & Akella, A. (2014, August). Multi-resource packing for cluster schedulers. *SIGCOMM Comput. Commun. Rev.*, *44*(4), 455–466. Retrieved from http://doi.acm.org/10.1145/2740070.2626334 doi: 10.1145/2740070.2626334

Gregg, B. (n.d.). *Linux Extended BPF (eBPF) Tracing Tools.* http://www.brendangregg.com/ebpf.html. (Accessed May 2020)

Gregg, B. (2019). *BPF Performance Tools.* Addison-Wesley Professional.

Hess, D. K., Safford, D. R., & Pooch, U. W. (1992). A Unix network protocol security study: Network Information Service.

Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., & Miller, D. (2018). The eXpress data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies* (pp. 54–66).

Ict statistics of 2001-2018. (2019). `https://www.itu.int/en/ITU-D/Statistics/Pages/stat/default.aspx`.

Internet growth statistics 1995 to 2019. (2020). `https://www.internetworldstats.com/emarketing.htm`.

In-band Network Telemetry (INT). (n.d.). (`https://p4.org/assets/INT-current-spec.pdf`)

Ioannidis, S., Anagnostakis, K. G., Ioannidis, J., & Keromytis, A. D. (2002). xpf: packet filtering for low-cost network monitoring. In *Workshop on high performance switching and routing, merging optical and ip technologie* (pp. 116–120).

Io visor project. (n.d.). `https://www.iovisor.org`.

Jacobson, V. (1997). *Pathchar: A tool to infer characteristics of Internet paths.*

Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., … others (2013). B4: Experience with a globally-deployed software defined wan. In *Acm sigcomm computer communication review* (Vol. 43, pp. 3–14).

Jeyakumar, V., Alizadeh, M., Geng, Y., Kim, C., & Mazières, D. (2014). Millions of little minions: Using packets for low latency network programming and visibility. *ACM SIGCOMM Computer Communication Review*, *44*(4), 3–14.

Jonathan, A., Chandra, A., & Weissman, J. (2018). Rethinking adaptability in wide-area stream processing systems. In *Proceedings of the 10th usenix conference on hot topics in cloud computing* (pp. 2–2). Berkeley, CA, USA: USENIX Association. Retrieved from `http://dl.acm.org/citation.cfm?id=3277180.3277182`

Karlsson, M., & Töpel, B. (2018). The path to dpdk speeds for af xdp. In *Linux plumbers conference.*

Kohler, E., Morris, R., Chen, B., Jannotti, J., & Kaashoek, M. F. (2000). The click modular router. *ACM Transactions on Computer Systems (TOCS)*, *18*(3), 263–297.

Kurose, J. (2017).

In *Computer networking: A top-down approach.*

Legtchenko, S., Williams, H., Razavi, K., Donnelly, A., Black, R., Douglas, A., . . . Rowstron, A. (2017, July). Understanding rack-scale disaggregated storage. In *9th USENIX workshop on hot topics in storage and file systems (hotstorage 17).* Santa Clara, CA: USENIX Association. Retrieved from https://www.usenix.org/conference/hotstorage17/program/presentation/legtchenko

Li, Y., Miao, R., Kim, C., & Yu, M. (2016, March). Flowradar: A better netflow for data centers. In *13th USENIX symposium on networked systems design and implementation (NSDI 16)* (pp. 311–324). Santa Clara, CA: USENIX Association. Retrieved from https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/li-yuliang

Linux cls_bpf. (n.d.). https://github.com/torvalds/linux/blob/master/net/sched/cls_bpf.c.

Linux kernel commit. (n.d.). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b4fc1a460f3017e958e6a8ea560ea0afd91bf6fe. (Accessed May 2020)

Linux SNMP Counter. (n.d.). (https://www.kernel.org/doc/html/latest/networking/snmp_counter.html)

Mars, J., & Tang, L. (2013). Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th annual international symposium on computer architecture* (pp. 619–630). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2485922.2485975 doi: 10.1145/2485922.2485975

Matteo Bertrone, M. V. B. (2016). Coupling the flexibility of OVN with the efficienty of IOVisor. In *Ovs conference.*

McCanne, S., & Jacobson, V. (1993). The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the usenix winter.*

Mogul, J. (1990). Efficient use of workstations for passive monitoring of local area networks. *ACM SIGCOMM Computer Communication Review*, *20*(4), 253–263.

Mogul, J., Rashid, R., & Accetta, M. (1987). The packet filter: an efficient mechanism for user-level network code. *ACM SIGOPS Operating Systems Review*, *21*(5), 39–51.

Narayanan, I., Wang, D., Jeon, M., Sharma, B., Caulfield, L., Sivasubramaniam, A., ... Vaid, K. (2016a). Ssd failures in datacenters: What? when? and why? In *Proceedings of the 9th acm international on systems and storage conference* (pp. 7:1–7:11). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2928275.2928278` doi: 10.1145/2928275.2928278

Narayanan, I., Wang, D., Jeon, M., Sharma, B., Caulfield, L., Sivasubramaniam, A., ... Vaid, K. (2016b). Ssd failures in datacenters: What? when? and why? In *Proceedings of the 9th acm international on systems and storage conference.* New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/2928275.2928278` doi: 10.1145/2928275.2928278

Netinet repository. (n.d.). `https://github.com/afabbro/netinet`.

Ntop's pf_ring zc (zero copy). (n.d.). `https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/`. (Accessed May 2020)

Ousterhout, A., Fried, J., Behrens, J., Belay, A., & Balakrishnan, H. (2019, February). Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX symposium on networked systems design and implementation (NSDI 19)* (pp. 361–378). Boston, MA: USENIX Association. Retrieved from `https://www.usenix.org/conference/nsdi19/presentation/ousterhout`

p4 on the edge. (n.d.). `https://schd.ws/hosted_files/2016p4workshop/1d/intel%20fastabend-p4%20on%20the%20edge.pdf`.

Paxson, V., Mahdavi, J., Adams, A., & Mathis, M. (1998). An architecture for large scale internet measurement. *IEEE Communications Magazine*, *36*(8), 48–54.

Pcap editing and replaying utilities. (n.d.). `https://tcpreplay.appneta.com/`, date = 1998.

Pelsser, C., Cittadini, L., Vissicchio, S., & Bush, R. (2013). From Paris to Tokyo: On the suitability of ping to measure latency. In *Proceedings of the 2013 conference on internet measurement conference* (pp. 427–432).

Peter, S., Li, J., Zhang, I., Ports, D. R., Woos, D., Krishnamurthy, A., ... Roscoe, T. (2016). Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)*, *33*(4), 11.

Popescu, D. A., Zilberman, N., & Moore, A. W. (2017). Characterizing the impact of network latency on cloud-based applications' performance.

Prasad, R., Jain, M., & Dovrolis, C. (2004). Effects of interrupt coalescence on network measurements. In *International workshop on passive and active network measurement* (pp. 247–256).

pyroute2 netlink library. (n.d.). `https://docs.pyroute2.org`. (Accessed May 2020)

Recap: High-performance Linux Monitoring with eBPF. (n.d.). `https://www.weave.works/blog/recap-high-performance-linux-monitoring-with-ebpf/`.

The rise of social media. (2020). `https://ourworldindata.org/rise-of-social-media`.

Rizzo, L. (2012). Netmap: a novel framework for fast packet i/o. In *21st usenix security symposium (usenix security 12)* (pp. 101–112).

Schulist, J., Borkmann, D., & Starovoitov, A. (n.d.). *Linux Socket Filtering aka Berkeley Packet Filter (BPF).* `https://www.kernel.org/doc/Documentation/networking/filter.txt`. (Accessed May 2020)

Snitch. (n.d.). (`http://cassandra.apache.org/doc/4.0/operating/snitch.html/`)

Sommers, J., & Barford, P. (2007, October). An active measurement system for shared environments. In *Proceedings of acm sigcomm internet measurement conference.*

Sundaresan, S., Deng, X., Feng, Y., Lee, D., & Dhamdhere, A. (2017). Challenges in inferring internet congestion using throughput measurements. In *Proceedings of the 2017 internet measurement conference* (pp. 43–56).

tcpdump and libpcap. (n.d.). `http://www.tcpdump.org/`. (Accessed May 2020)

Tennenhouse, D. L., Smith, J. M., Sincoskie, W. D., Wetherall, D. J., & Minden, G. J. (1997). A survey of active network research. *IEEE communications Magazine*, *35*(1), 80–86.

Thacker, C. P., MacCreight, E., & Lampson, B. W. (1979). *Alto: A personal computer.* Xerox, Palo Alto Research Center Palo Alto.

A thorough introduction to eBPF. (n.d.). https://lwn.net/Articles/740157/. (Accessed May 2020)

Toonk, A. (2020, April). *Building an XDP (eXpress Data Path) based BGP peering router.* https://medium.com/swlh/building-a-xdp-express-data-path-based-peering-router-20db4995da66.

Tran, V.-H., & Bonaventure, O. (2019). Beyond socket options: making the linux tcp stack truly extensible. In *2019 ifip networking conference (ifip networking)* (pp. 1–9).

Tu, W. C.-C. (2016). Offloading OVS Flow Processing using eBPF. In *Ovs conference.*

Turull, D., Sjödin, P., & Olsson, R. (2016). Pktgen: Measuring performance on high speed networks. *Computer communications*, *82*, 39–48.

Using eBPF for network traffic analysis. (2018). https://www.ntop.org/wp-content/uploads/2018/10/Sabella.pdf.

Von Eicken, T., Basu, A., Buch, V., & Vogels, W. (1995). U-net: A user-level network interface for parallel and distributed computing. *ACM SIGOPS Operating Systems Review*, *29*(5), 40–53.

Wirtgen, T. (2019). *Leveraging ebpf to improve the flexibility of routing protocols: the case for border gateway protocol (bgp)* (Master's thesis, Ecole polytechnique de Louvain, Université catholique de Louvain). http://hdl.handle.net/2078.1/thesis:19606.

Wu, Z., Xie, M., & Wang, H. (2008). Swift: A fast dynamic packet filter. In *Nsdi* (Vol. 8, pp. 279–292).

Wu, Z., Xie, M., & Wang, H. (2011). Design and implementation of a fast dynamic packet filter. *IEEE/ACM Transactions on Networking*, *19*(5), 1405–1419.

XDP-Project. (n.d.). https://github.com/xdp-project. (Accessed May 2020)

Xhonneux, M., & Bonaventure, O. (2018). Flexible failure detection and fast reroute using ebpf and srv6. In *2018 14th international conference on network and service management (cnsm)* (pp. 408–413).

Xhonneux, M., Duchene, F., & Bonaventure, O. (2018). Leveraging ebpf for programmable network functions with ipv6 segment routing. In *Proceedings of the 14th international conference on emerging networking experiments and technologies* (pp. 67–72).

Yu, M., Jose, L., & Miao, R. (2013). Software defined traffic measurement with opensketch. In *Presented as part of the 10th USENIX symposium on networked systems design and implementation (NSDI 13)* (pp. 29–42). Lombard, IL: USENIX. Retrieved from `https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/yu`

Ziviani, A., Cardozo, T. B., & Gomes, A. T. A. (2012). Rapid prototyping of active measurement tools. *Computer Networks*, *56*(2), 870–883.