NRG: A Network Perspective on Applications' Performance

Noa Zilberman
University of Oxford

Andrew W Moore
University of Cambridge

Billy Cooper
Unaffiliated

Jackson Woodruff
University of Edinburgh

Yuta Tokusashi
University of Cambridge

Pietro Bressana
Universitá della Svizzera italiana

Murali Ramanujam
UCLA

Diana Andreea Popescu
University of Cambridge

Salvator Galea
University of Cambridge

***Abstract*—Running an application in a data center, users have an important goal: performance. Even though mismatched application requirements and network resource assignments can lead to significant performance loss, the understanding of dynamic networking effects on data center applications' performance is still limited. In this paper we present *NRG*, an open source toolset that enables reproducible experimentation and provides a networking perspective on applications' performance. NRG is a portable and programmable solution operating at line rate, and enabling users to recreate data center network conditions as a black box within controlled, small-scale experimentation environments. We demonstrate the potential of NRG to solve performance issues and provide insights on several applications.**

***Index Terms*—Reproducibility, Data Center, Performance**

I. Introduction

Cloud computing is an appealing environment for many users: it is relatively cheap, does not require maintenance by the user, promises availability, and provides performance that is better than most users can achieve on their local systems. The data centers enabling these cloud environments are driven by performance and cost, trying to appeal to end users and aspects of application performance within data centers have been extensively studied (e.g., [1], [2]).

Network performance within the data center can be defined in many different ways: one application may target throughput as a performance metric, while another might prefer tail latency as the main performance goal. But the mapping between network performance metrics and application performance metrics such as cost, power efficiency, and task completion time is often not clear.

In this paper, we assert that improving application performance requires understanding application interaction with the underlying infrastructure, and as is the focus of this work: the network. Without understanding which applications are network intensive and which are network agnostic, data center resource allocation cannot be optimal. A mismatch between application requirements and network resources can lead to reduced application performance and affect resource utilization [3]. Unfortunately, the data center network is a black box, obscured from the user, and it is hard to infer the network's effects on an application, let alone in a reproducible manner. Small-scale experiments can provide the observability and reproducibility required, but lack realistic data center network conditions, meaning performance can remain a mystery.

We introduce NRG, a Network Research Gadget (pronounced *en-er-gy*). NRG enables reproducible networking experimentation through network emulation and monitoring — recreating realistic data center network conditions in small-scale environments. NRG is an open-source hardware-software toolset with bandwidth and nanosecond-scale latency and jitter control, providing a black box representation of a data center network. NRG also provides a programmable, hardware accelerated, line-rate monitoring frameworks that offloads information processing from the end-host to the network and alleviates the need for frequent network probing. NRG can be either standalone or a core within a network device. We prototype NRG on NetFPGA SUME [4], port it to two more FPGA platforms, and partially to Intel Tofino ASIC.

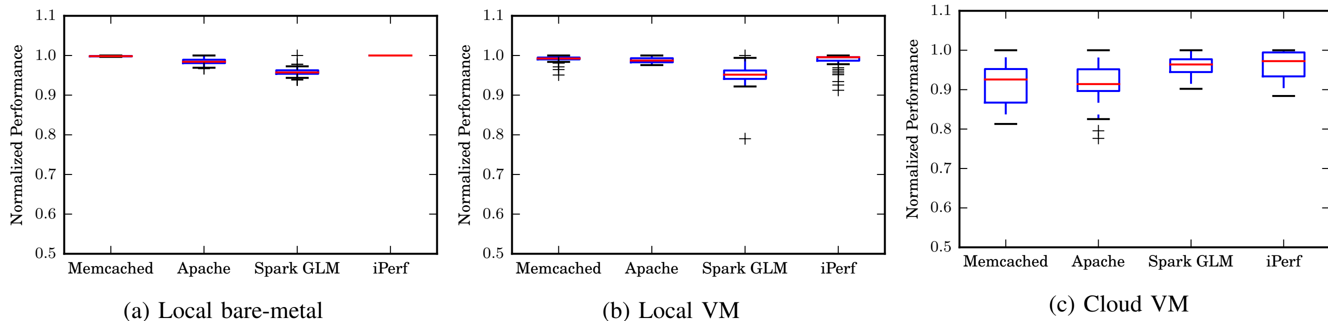
We use NRG to generate *Network Profiles*, the combination of an application's performance and the characteristics of the underlying network. NRG considers high-level application performance metrics (e.g., queries per second) over network performance metrics (e.g., flow completion time), allowing direct understanding of how network configuration affects application performance. Using two case studies, we demonstrate that network profiles unveil application performance problems and enable better network provisioning.

In summary, this paper makes the following contributions:

- We introduce NRG, a hardware-software toolset for reproducible networking research.
- We introduce the concept of *Network Profiles* to characterize an application's performance based on network characteristics, and a generation methodology using NRG.
- We describe a case-study using NRG to understand the performance of four typical data center applications.
- We describe a case-study using NRG to debug users' performance issues, and show the importance of the nanosecond and microsecond scale monitoring enabled by NRG.

II. Motivation

Determining the best provision for a cloud application can be challenging. Cloud computing provides compute resource choices, from the type of virtual machine (VM) to the CPU.



(a) Local bare-metal

(b) Local VM

(c) Cloud VM

Fig. 1: Normalized performance for applications operating on (a) bare-metal local machines, (b) within a VM on an otherwise quiescent local machine, and (c) within a VM in the cloud.

Networking resource selection is limited, and CPU or VM-class often dictates available bandwidth and sometimes latency. Even if an application's network properties are known, it is hard to infer the optimum choice of cloud resources.

Once a cloud application is deployed, it is difficult to assess the weight of different factors on the overall application's performance. If a change in an application's code unintentionally affects performance, it is hard to find and debug it, even if the effect is at the scale of tens of percentages.

To illustrate the challenge, we compare applications' performance in three different environments: a local (self-controlled) data center, VMs within local data center, and a cloud environment (Azure). In each experiment we set two machines, e.g., one as a client and one as a server, and run an application benchmark 100 times. There are no other running workloads or cross traffic in the local data center experiments. In the cloud environment, the VMs are set within the same data center. The applications we choose are Memcached (key-value store), Apache (web server), Spark GLM (machine learning) and iperf (throughput test). The applications are further described in our artifact [5]. Figure 1 illustrates the performance variance per application, where the performance metric varies between applications (e.g., training time, requests/second). The maximum performance achieved is normalized as 1.0, and the change in performance is over 100 runs.

As Figure 1 shows, the differences between running on a local machine, either (a) on bare-metal or (b) on a VM, and (c) in a cloud environment are large. Locally, the differences between the 25th and 75th percentiles are negligible for Memcached, Apache and iperf, and around 2% for Spark. In contrast, in the cloud environment, there can be over 20% difference in performance between runs, and between the 25th and 75th percentiles the differences in performance can reach 8%. These results demonstrate clearly that it is hard to benchmark performance improvement of cloud applications, and to validate performance-related code changes.

Our experiment does not show that the network is the cause of variance. It does demonstrate, however, the difficulty of running reproducible experimentation in the cloud, especially where the goal is applications' performance benchmarking.

Any approach to reproduction of a system exists on a spectrum of precision, accuracy, repeatability, correctness and so-forth. Balancing these allows many choices, from simulator to simplified test, and from emulation to a complete environment. Emulation environments such as Mininet [6] offer many desirable properties, but trade-in functionality with performance and complexity. Simulators such as ns-2 and ns-3, enable repeatability in results, yet have known drawbacks: often limited in completeness, unconstrained by CPU or memory usage in the same way an actual implementation would be, and bounded by extremely long simulation times.

Software-based emulation tools such as DummyNet [7] and NetEm [8] fail to emulate the network both at high data rates and at microsecond-level latency [9]. To illustrate through an experiment, in Figure 2 we measure the accuracy of microsecond-scale delay imposed by NetEm and NRG on a packet, using the setup described in §V-B and in [5]. As the figure shows, the delay error (y-axis) increases as latency (x-axis) decreases. For 1μs delay, the median latency imposed by NetEm may be up to five times higher than requested. In contrast, NRG provides nanosecond-scale precision. Microsecond scale latency is increasingly important in data centers [2], with VM-to-VM latency being on few microseconds scale. Given the sensitivity of applications to microsecond scale latency [3], we need to be able to emulate network latency, using higher resolution and precision than provided by software emulation.

In NRG, we focus on a subset of these challenges. NRG provides a realistic local evaluation environment that can provide each and every time the same (network) test conditions. Coupled with that, NRG also provides an emulation of the data center network as a "black box", in terms of latency and bandwidth. NRG provides nanosecond-resolution control over latency combined with support of line-rate traffic. The data rate supported by NRG is the same as an instantiated network device (NIC, switch or bump-in-the-wire), and seamlessly scales with it. By changing configuration seeds, NRG enables creating similar-but-not-identical scenarios, yet still repeatable, enabling users to explore a wider range of "what can go wrong" scenarios. Beyond that, NRG depends on the user's network and not on the user's hosts, and is not constrained by CPU resource limitations.

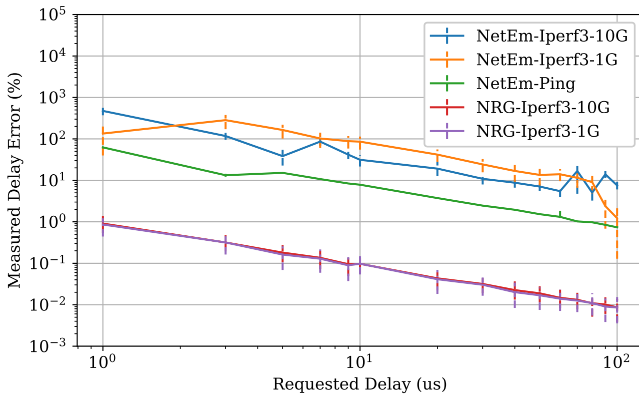


Fig. 2: The measured error of median delay under various loads, using NetEm and NRG. NetEm introduces up to 500% error for a microsecond delay, whereas NRG error is two orders of magnitude smaller, in the order of nanoseconds.

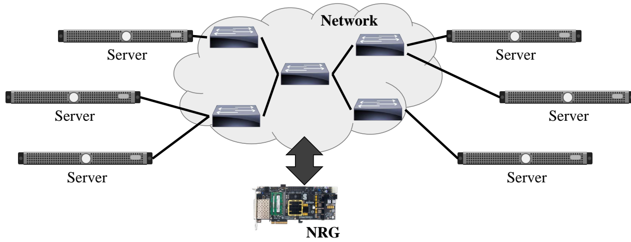


Fig. 3: A cloud-based black box experimentation model. NRG replaces part or all of the network black box.

III. Network Profiles

Small network changes may produce large performance changes. **Network Profiles** refer to a collection of network-related characteristics and their relation to an application's performance. Network profiles are designed for users with an application to deploy and a given system setup (e.g., servers, NICs, operating system) but who wish to know:

- Is my application bottlenecked by the network?
- Is the network well utilized?
- Can network changes improve applications' performance?

NRG allows to run on a **local** setup experiments that actively change network conditions (bandwidth, latency, burst size) and passively collect network statistics (e.g., inter-packet gap, link utilization, flow size). This collection of network conditions and statistics is the **Network Profile** of an application on a given setup.

While the uncontrolled black box cloud network behavior cannot be predicted, it can be measured [10], [11]. NRG can use these measurements to reproduce the cloud's network behavior, by recreating similar latency (nanosecond-scale), bandwidth (Mbps resolution) and burstiness (byte-scale) scenarios. The operational model of NRG is the substitution of an uncontrolled cloud-based black box experimentation model with a local user setup incorporating NRG, as shown in Figure 3. NRG can act as a bump in the wire or as a *transparent* module within a NIC or switch. Wherever NRG is instantiated, it can programmatically limit available bandwidth and increase latency for a given link. This, when combined with other configuration parameters for network: e.g., link MTU, and application: e.g., process and thread configuration, results in performance measurements along a multidimensional surface representing the relationship between performance and the control parameters.

Figure 4 illustrates network profiles in 3D for a number of applications; for a given configuration, it illustrates the relationship between bandwidth and static latency on application performance. This case study is discussed in § VII.

IV. NRG Architecture

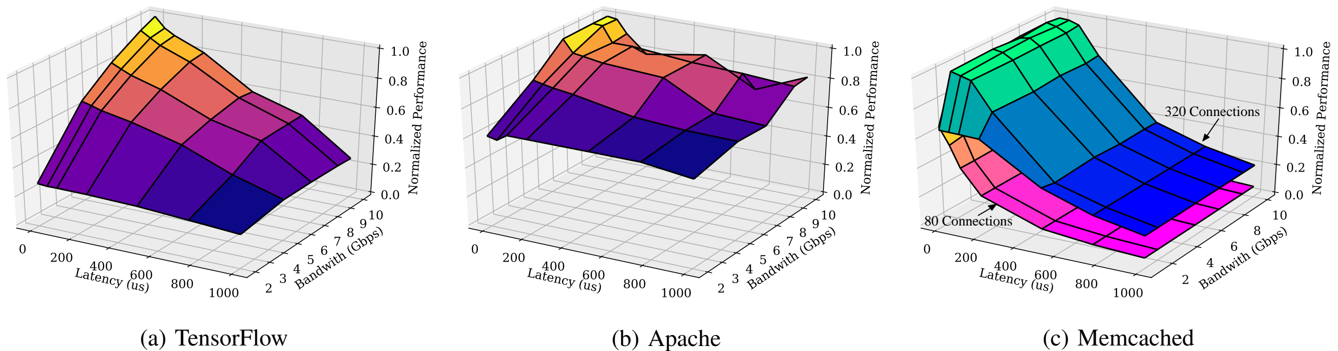
NRG is a hardware/software toolset permitting fine-grained control and measurement of network characteristics for reproducible networked-systems research. NRG works in a controlled experimentation environment. NRG combines a set of properties required for reproducible experimentation:

- A software module for experiment orchestration, configuration and control of multiple nodes in a system.
- A hardware module, emulating the network black box, on nanosecond resolution and at line rate.
- A hardware module, monitoring the network and processing collected information at line rate.
- A software module, collecting results from all nodes and hardware modules, and generating network profiles.

An experiment begins by configuring a set of NRG-enabled devices within a networked-system. Once the setup is ready, the NRG's orchestration module triggers the experiment. At the end of the experiment, the application's results are collected. Monitoring information is also collected from the system and from NRG-enabled devices. Last, the collected information is processed and network profiles are generated.

Control and Orchestration: NRG targets networked systems of multiple nodes (servers) and allows multiple NRG devices. A single control and orchestration node sets experiments on all the participating nodes, by installing applications and setting system configurations. The same node sets up other NRG-enabled devices, including platform configuration and loading software modules.

Network Emulation: Network emulation is an in-band component of NRG. It can be a stand-alone bump-in-the-wire device, or part of a more complex device (e.g., switch).Figure 5 illustrates a typical NRG-enabled device's architecture. The in-band network emulation module provides latency control (delay module), and bandwidth and burstiness control (rate control module). The emulation module is instantiated after the data-plane enabling fine-grain functionality, e.g., applying delay to specific flows indicated by the data plane. The delay module inserts latency either as a constant, or from a distribution. Constant latency is equivalent to adapting the distance between machines. Latency distributions describe a combination of static and variable latency, using either custom or pre-defined distributions (e.g., uniform, normal, pareto and pareto-normal). Delay granularity is on a nanosecond time scale and depends on the target platform.



(a) TensorFlow (b) Apache (c) Memcached

Fig. 4: Normalized performance of several applications when subject to a range of constrained bandwidths and static latencies. Experiments for these figures are discussed in § VI.

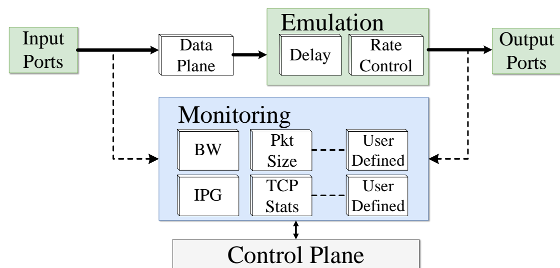


Fig. 5: An illustration of NRG hardware architecture. Data plane and control plane connections are omitted for brevity.

Network Monitoring: NRG uses hardware acceleration to provide network insights with no host processing. NRG's monitoring is passive, instantiated in parallel to the data plane. The module monitors properties such as link utilization, inter packet gap, user defined statistics and more. The module is triggered at the beginning of an experiment, and logs statistics only upon events, avoiding unnecessary resource consumption. Instead of saving individual counters, NRG automatically generates (probability) distributions of monitored events. The key insight for the distribution generation is; for most research and monitoring purposes, some accuracy can be sacrificed through binning to achieve feasibility.

Data Collection and Processing: At the end of an experiment, a software module collects instrumentation information from all nodes to a create network profile. This includes performance results from end hosts, statistics from NRG, etc. It also collects reproducibility data about the system under test. The module's second role is processing results aggregated across experiments, e.g., across a range of latencies. The data from the monitoring module can present insights, e.g., throughput as a function of burst size.

A. Network Emulation Architecture

NRG's network emulation is divided between latency and rate control. Latency control is designed as a queue (FIFO) with a release mechanism. Every packet that enters NRG is timestamped (t), and assigned a delay d. The delay d is composed of a static latency component and a random latency component, chosen from a (programmed) latency distribution. The packet is released from the queue whenever the current time is bigger than $t + d$.

To support variable latency, NRG uses small on-chip memories (1K-8K entries) that store distributions, and a pseudo random bit sequence (PRBS) generator that points to random addresses within the memories. The depth of the memory determines the granularity of the distribution. The distribution memory can be populated by the user, or a predefined distribution can be selected. While the memories define the *shape* of the latency distribution, the *scale* of the distribution is defined through a multiplier configuration.

NRG's queue design does not enable reordering, similar to packets traversing through an identical path. Reordering is possible between packets of different flows, where some packets go through the delay module while others don't.

Rate control is a common function in network devices. NRG implements it as a FIFO (queue) with a release mechanism which dictates both bandwidth and burst size. The module is primarily designed to imitate a lower-speed link. As such, it effectively reduces the progression speed through the pipeline, and works on bus-width (byte) level granularity. Typically, the rate control module will be located after the latency module, and use a FIFO that is an order of magnitude smaller than the latency queue. Propagating flow control allows to account for queueing effects within the rate control module when discharging packets in the latency module.

V. NRG Implementation

The NRG implementation is open source, available at [5]. It was prototyped on multiple FPGA platforms, and is open to community contributions. Already, NRG has been used in several projects (§VI), and has proved portable and useful. The implementations described below are the ones released with this project, but users can easily adapt the tool to their environment and needs. We explore in §V-D design portability.

A. Prototype

The control, orchestration and performance monitoring software components are implemented in Python, using C to interact with the NRG. The prototype supports scripting environments, and a GUI for manual configuration and testing. Test

setup and application configuration are taken from files. Each test can sweep ranges of network parameters (e.g., latency from 0 to 100μs in 1μs steps using jitter). Tests produce logs and generate performance graphs.

NRG was first prototyped and evaluated on $4 \times 10GE$ NetFPGA SUME [4]. The platform allows 5ns latency control resolution, and bandwidth of 1Mbps to 10Gbps per port. This is a standalone bump-in-the-wire design, supporting two ports (for maximum latency scalability). It does not change relations between competing flows. The monitoring module supports monitoring per port, pattern match or of a flow. Multiple monitoring mini-blocks are implemented, such as link utilization and packet rate, TCP window size and flow size. The monitoring module consumes few resources; just 0.62% of logic and 2.2% of memory resources.

B. Validation

We validate our NRG prototype's functionality and performance using OSNT [12] as a traffic generator and Cisco Nexus NIC HPT for packet capture, using an optical splitter. The latency accuracy of NRG is $\pm$30ns independent of the configured static latency, similar to a NetFPGA reference design. For latency distribution, we sweep the parameters of both distribution values, and scale of distribution (i.e., from nanoseconds to hundreds of microseconds), and use Kolmogorov-Smirnov to test fit, and visually compare expected and actual distributions. Rate control is validated from zero to 10Gbps.

Monitoring is evaluated using trace capture, comparing the properties of captured traffic to NRG statistics. The accuracy is capped by the level of data aggregation (e.g., 1ns, 1μs, 1ms).

The validation shows that NRG works to its specification. It is not compared with a production data center network, nor aims to fully recreate a data center environment.

C. Programmability

NRG enables drop-in custom monitoring mini-blocks. These are not language bound, and we have modules coded in Verilog, P4 and .NET. We used P4-NetFPGA [13] for a P4-based implementation, including statistics such as bandwidth, packet size distribution, and inter-arrival time distribution. Our implementation of user statistics in .NET used Emu [14], for example for monitoring packet reordering on a port and flow level, and for bandwidth statistics.

NRG's emulation active path, the delay and rate control modules, are implemented in Verilog. Such modules are less suitable for high-level languages (e.g., P4), except as externs.

D. Portability

FPGA: FPGA targets can operate as a bump-in-the-wire, as a NIC and as a limited-size switch. NRG was ported to two FPGA targets: Xilinx VCU1525 (XCVU9P FPGA) and U280 (XCU280 FPGA), each with $2\times100G$ ports. Running on these FPGA targets provides higher resolution than on NetFPGA SUME (4ns vs. 5ns), more on-chip memory (35MB/41MB vs. 6.5MB) and different network interfaces (100G vs. 10G).

SmartNIC: SmartNIC portability depends on the NIC's architecture (ASIC, FPGA or SoC based). NRG can be fully ported to FPGA-based smartNICs. For other types, portability depends on the level of programmability.

Switches: We explored porting NRG's monitoring to two switch ASICs: Barefoot Tofino and Broadcom Jericho 2. Our observations are based on coding (Tofino) and discussions with both ASIC teams. In Tofino, the monitoring design could be ported with some changes, e.g., using Tofino's built-in externs, timestamp counter and registers. The Jericho 2 platform can support monitoring, but requires porting the code to C++. Most high-end switches already support rate control with fine granularity. Latency control, on the other hand, is less feasible.

E. Scalability

Many NRG devices can be used within a system, and each device is independent from the others. Device setup is asynchronous, and hardware mechanisms enable starting on a triggered event (e.g., first packet of a certain type), or on a configuration trigger. This assists in distributed systems and does not preclude synchronized operation.

NRG also scales with data rate. Our 10G prototype and 100G prototypes use similar libraries and can continue and scale with port rate. If we consider the monitoring functionality, NRG will process 150Mpps at the same ease that it processes 15Mpps. This is significantly different to the resources required by host-terminated monitoring application. Latency-wise, while the number of packets NRG can buffer does not change with data rate, if data rate increases, the maximal latency supported at full line rate decreases. Therefore, faster port rates require deeper memories.

VI. Use Cases

NRG was developed for exploring, understanding and reproducing networked experiments. In this section, we describe a few potential use cases of NRG and network profiles.

Reproducibility: NRG enables reproducible experiments, while varying network conditions and emulating a cloud network as a black box. It enables a stable experimental environment that recreates network conditions identically between experiments, and allows recreation of failure conditions. Latency distribution seeding allows reproducible creation of a range of congestion scenarios. NRG enables repeatable benchmarking and comparison of solutions. We envision an NRG-enabled artifact evaluation environment, to test research artifacts prior to publication, and providing reliable performance comparisons between solutions under different scenarios.

Resource Allocation: Understanding distributed application performance is hard. NRG enables studying the sensitivities of applications to bandwidth and latency through experimentation in controlled environments. Using network profiles, resource allocation can be improved, such as the number of clients and their link capacity.

Understanding and Debugging Applications' Phenomena: Network Profiles provide a better understanding of applications' performance, providing insights beyond a single resource. As we show in § VIII, when performance changes,

NRG enables debugging: is it due to congestion or high network utilization? Has the burstiness changed? Is the network responsible at all, or was network behavior unchanged? Similarly, NRG enables a software development loop with network-level application behavior.

VII. Case Study: Exploring the Effects of the Network on Performance

It is known that bandwidth and bandwidth variability affect performance [15], and that some applications are sensitive to sub-millisecond latency [3]. In this case study, we use NRG to explore in a reproducible manner how choices of network resources affect the performance of different applications. This is done for a small-scale setup (24 cores), comparable with the default reserved instances quota in some cloud services [16].

A. Setup and Applications

We use an experimental setup [5] in our local data center composed of 6 hosts. Each host has an Intel Xeon E5-2637 v4 CPU with four cores (24 total cores), running at 3.5GHz with 64GB RAM. The hosts run Ubuntu Server 16.04, kernel version 4.4.0-131-generic, using default network configurations (e.g., TCP Cubic). Each host is equipped with an Intel X520 NIC, connected at 10Gbps to an Arista 7124FX switch. The median round-trip time (RTT) between clients and server is 10μs, and only one workload is running at a time (no other cross-traffic). We use one host as a server and five as client machines, and instantiate an NRG device between the server and the switch, acting as a port-level bump-in-the-wire.

We use several popular applications. Their choice explores increasing application complexity and a number of operating models. We overtly prefer network intensive applications. The simplest application is a domain name server (DNS, measured in requests per second). We also benchmark Apache webserver (measured in requests per second). We benchmark Memcached, a key-value store application, measured in queries per second. As a workload, we use *Facebook "ETC"* [17]. Last, we pick TensorFlow, a machine learning framework, and use the MNIST dataset for training using distributed learning, measured with training time.

We study the effect of bandwidth and static latency on application performance, varying these parameters between the server and switch. In this way, we emulate a VM with different bandwidth allocation and different data center location. This experiment explores the effect of these parameters alone on the end-to-end performance, and no other perturbations. Each experiment is run ten times. We generate a network profile for each application, also containing the variance. Below, we explore dependencies between application network presence (e.g., link utilization, packet size) and resource allocation.

The scale of latency may vary significantly between cloud providers, and over time. The latencies used in our experiments match previous works [18]. We validated our latency scale in Azure (using AccelNet). We found [5] latencies on the order of 128μs-173μs minimum latency and 300μs-500μs median latency (for different machines).

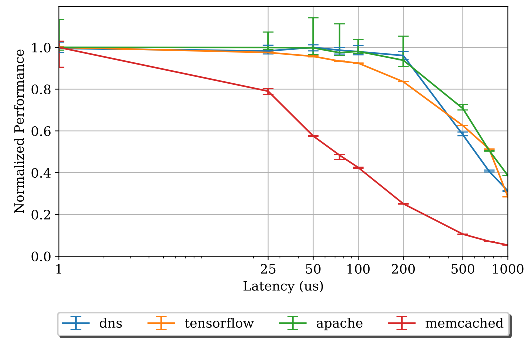


Fig. 6: The effect of static latency on different applications.

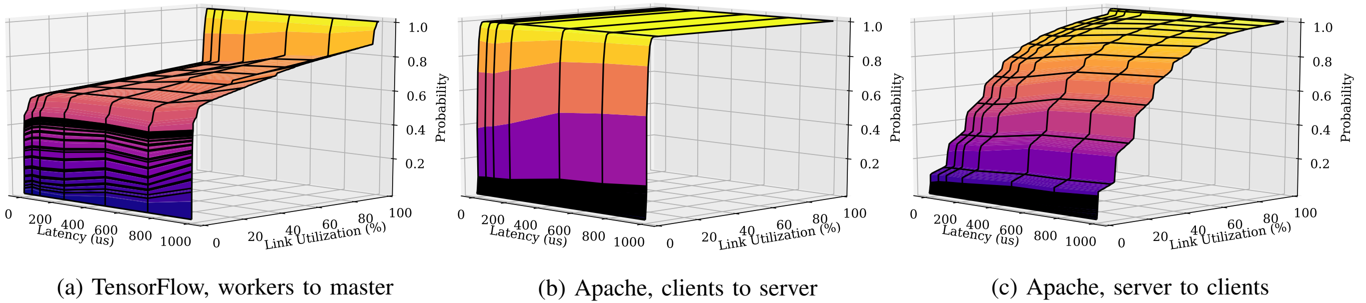
B. Experimental Results

Figure 6 compares the effect of static latency on all four applications. The times indicate one-way latency and are applied in both directions, similar to distancing a server from a switch. The switch-server bandwidth is 10Gbps. Memcached is extremely sensitive to latency, losing 20% performance with 25μs added, and 58% when 100μs are added. TensorFlow loses 2.5% performance with 25μs added, and 8.3% with 100μs. At 500μs, or 1ms RTT, all applications lose between 29% (Apache) and 90% (Memcached) performance. While sensitivity to latency is not new, these results show the magnitude of the effect even on very short timescale.

Next, we vary both bandwidth and latency and explore the performance effects (Figure 4). TensorFlow (Figure 4(a)) is almost linearly sensitive to latency and bandwidth. There is little variability between experiments: $<$1.5% across all scenarios. We cannot determine whether TensorFlow is "more sensitive" to bandwidth or latency. However, we can say, for example, that (on our setup) TensorFlow with 10G bandwidth and 100μs of added RTT, will perform slightly worse than when allocated 9Gbps bandwidth and no added latency. The monitored network properties (bandwidth utilization, packet rate, burst size and inter-packet gap) of TensorFlow on server's transmit and receive side are similar. Other applications show significant differences between transmit and receive directions.

One component of TensorFlow's performance profile is link utilization. Figure 7(a) shows client-to-server bandwidth utilization of TensorFlow, with bandwidth of 10Gbps and varied latencies. As shown, the link is idle more than half the time, and 20% of the time there is no traffic at all. Despite that, when no latency is added, TensorFlow utilizes more than 95% of the link for 26.5% of the time. With a millisecond RTT, the link achieves over 95% utilization for 19.4% of the time. Therefore, TensorFlow will work best with closely placed VMs and very high bandwidth links. We explore this further in [19].

Apache presents a more complex picture (Figure 4(b)). The performance with bandwidth of 8–10Gbps or 0–100μs RTT almost does not vary. The performance drops by 10% with 8Gbps of bandwidth and 100μs RTT, or if the RTT is 200μs. If bandwidth is further reduced, it dominates latency in performance loss, meaning it is better to have 10Gbps link



(a) TensorFlow, workers to master (b) Apache, clients to server (c) Apache, server to clients

Fig. 7: Bandwidth utilization of TensorFlow and Apache. For every latency configuration (x-axis) the graph shows the probability (z-axis) of a certain link utilization (y-axis), creating a surface of CDFs.

with millisecond latency, than to have a low-latency 2Gbps link. This is unsurprising given the file transfer nature of Apache web server. Figures 7(b) and (c) show Apache's link utilization in the client-to-server and server-to-client directions with 10Gbps bandwidth. Requests utilize little bandwidth, whereas replies utilize most of the link, though not as much as TensorFlow. As latency increases, Apache's link utilization decreases, matching the results in Figure 4.

Memcached (Figure 4(c)) is an interesting case. Our initial experiment, using typical 80 connections, showed high latency sensitivity, as also shown in Figure 6, but low sensitivity to bandwidth, with just 7% performance drop at 2Gbps. These results are the bottom surface shown in Figure 4(c). Consequently, we reduced bandwidth to 1Gbps, where throughput dropped by 35%. Even at 1Gbps latency is dominant, meaning that Memcached with 100μs RTT added has almost identical performance between 1–10Gbps bandwidth.

While we saturated server performance in our initial setup, as a second experiment we increased the number of connections per client thread $\times 4$, to a total of 320 connections per server (upper surface in Figure 4(c)). The bandwidth sensitivity remains unchanged, but latency resilience is improved, as the performance only slightly changes up to 200μs, and with a smaller performance drop at higher latencies.

For DNS, link utilization was extremely low, under 1Gbps (see [5]). We found it is server-bound and comparable to previous works (e.g., [14]).

Investigating burstiness, TensorFlow maximum burst size scales from 53 packets without added latency, to over 4K packets with 1ms delay[1] (Figure 8). In contrast, for all other applications the maximum burst declines with latency, e.g., from 25 to seven in Memcached and from 27 to 16 in Apache.

We explore variance between experiments. For Memcached, it is always $<$0.1%, and for TensorFlow it is $<$1.5% and typically $<$1%. The only application with significant variance is Apache, with up to 18% variance. This variance decreases with the number of clients.

C. Discussion

Our experience with TensorFlow reflects the application design in synchronous training mode, with bursty data distribution and gradients aggregation. The magnitude of effect

[1]Two packets are considered a burst if within 100ns from each other.

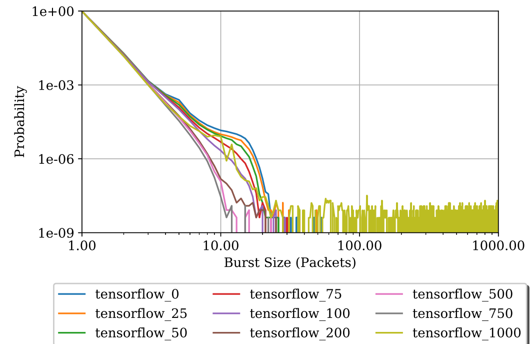


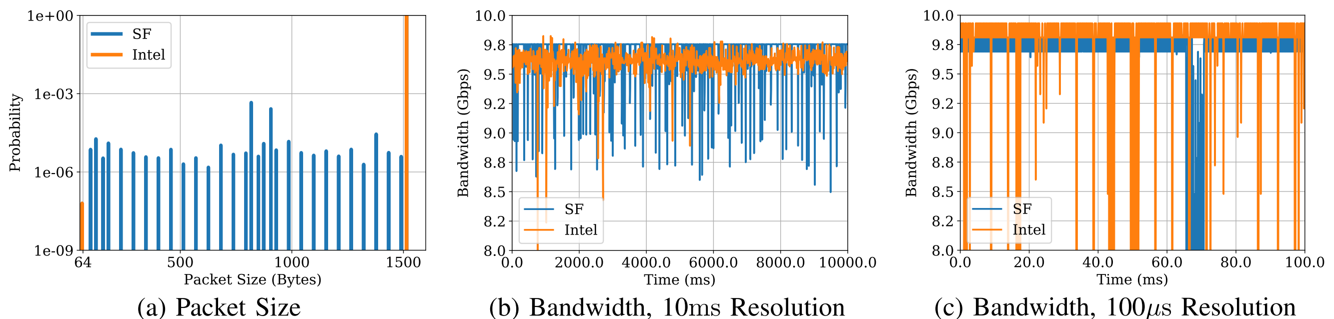
Fig. 8: TensorFlow burstsize probability distribution as function of latency, master to workers. Maximum burst size increases with latency. Colors indicate different latencies in μs.

for small link-bandwidth variances and utilization is important, as communication dominates some model's training time [20]. Applications such as Apache and TensorFlow improved as we added clients and workers, but Memcached is less affected. Memcached has a known network-related bottleneck due to request processing overhead relative to payload. This led Facebook to batch multiple requests within packets [21].

NRG's network profiles can provide valuable insights for switch vendors and cloud operators looking to reduce network congestion, by turning deployed programmable network devices to NRG-enabled. For example, monitoring burst size information is useful to assess buffer sizing [19]. NRG also indicates if bursts are *frequent*, which is beyond watermarks or telemetry packets [22].

VIII. Case Study: Debugging Performance Issues

We demonstrate how NRG can aid practical debugging, in this case a performance issue in the NetFPGA community. A NetFPGA SUME Reference Switch is known to run full line rate of $4 \times 10G$ ports. A NetFPGA community member reported *iperf3* achieving a bandwidth of 8.73Gbps with multiple retransmissions reported, while a test in the NetFPGA regression environment resulted in 9.30Gbps and no retransmissions. The difference was pinpointed to the NICs: Intel 82599ES had the issue, while Solarflare SFN6122 and SFC920 did not, running on the same setup. Connecting two Solarflare NICs directly resulted in the same iperf performance as with the Reference Switch in-between, while connecting



(a) Packet Size (b) Bandwidth, 10ms Resolution (c) Bandwidth, 100μs Resolution

Fig. 9: Using NRG to compare Iperf3 network profile on Solarflare (SF) and Intel NICs: (a) Packet size distribution (b) 1K bandwidth samples, 10ms resolution (c) 1K bandwidth samples, 100μs resolution.

two Intel NICs directly resulted in higher performance than with the switch in-between, and no retransmissions. Tweaking NIC configurations (e.g., rx-usecs) improved the performance but did not eliminate the problem. The user had limited access to debug and capture equipment.

We turned the user's switch into an NRG-enabled design by instantiating NRG's monitoring core. The user repeated the tests with NRG. An immediate high-level observation using network profiles was that the two NICs differ in packet sizes distribution, shown in Figure 9(a): while Intel NIC sends only 1518B packets, Solarflare's packet sizes range from 102B to 1518B[2]. We also observe packet size difference in the other direction (server to client): 70B on Solarflare vs 64B on Intel. This packet size difference is also observed with other versions of iperf. The IPG (inter-packet gap) varies between the two NICS: for the Intel NIC there is a single IPG of 3 clock cycles, and 88.5% of the IPGs are 57 to 60 clock cycles or more. For the Solarflare NIC, 85.7% of the packets have an IPG of 60 to 64 clock cycles, with 0.08% having a shorter IPG.

These observations do not explain the Intel NIC's performance, but did indicate different dynamics within the NetFPGA pipeline. Observing the switch input bandwidth has provided a further insight: iperf reports the bandwidth using a resolution of seconds. Using measurements taken over 10ms (Figure 9(b), showing 1K samples) showed the median bandwidth to be around 9.61Gbps on the Intel NIC and 9.75Gbps on the Solarflare NIC. A higher measurement resolution of 100μs (Figure 9(c), showing 1K samples) showed the median momentary bandwidth for the Intel NIC is 9.81Gbps and 9.77Gbps for Solarflare, an effect of the packet size distribution. Once identified that the Intel NIC had a higher momentary bandwidth than Solarflare, but with occasional drops, we tracked the output bandwidth from the device and identified flow control asserted by the output port of NetFPGA SUME. The root cause of the performance issue was tracked to transmit port inefficiency with the NetFPGA design, which led to the flow control. Fixing this inefficiency solved the problem.

The high-resolution of NRG's hardware-based bandwidth measurements helps identify such problems and correct them. This case study has inspired additional enhancements to NRG, such as a link utilization mini-block.

[2]We observe a single 64B IPv4 Packet during both NIC tests

IX. Related Work

Improving the performance of applications in the data center has been the focus of many works (e.g., [1], [23]), with significant focus on the contribution of the network to performance (e.g., [15], [24]). While previous work (e.g., [25]) considered application performance with the latency between the user and the data center, our work focuses on latency within the data center, meaning latencies of orders of magnitude lower.

Latency: Network latency has been identified long ago as a crucial factor for a good user experience [26]. Today, microsecond-scale latencies are considered the hardest to control [2], [3]. As tail latencies are considered an issue for interactive data center applications [27], several works have been proposed in this space (e.g., [28], [29]).

Monitoring Tools: Many monitoring tools are being used in the cloud [30], for purposes ranging from SLA management to troubleshooting. As NRG is research-focused rather than e.g., for billing, it allows for some loss of data in return for longer observation periods. Contrary to other hardware-based tools [31]–[33], NRG does the processing in the hardware, without sending packets to the host. Solutions such as Marple [31] can be leveraged to query NRG's monitoring.

Reproducibility: Networking research is hard to make repeatable or reproducible [6]. The research community is increasingly aware of the challenge and is encouraging its members to contribute published artifacts [34]. In addition to works introduced in earlier sections, approaches to reproducibility range from simulation (e.g., ns-2, ns-3), through emulation [6] and test beds to (relatively) small-scale full reproduction of a test environment. DataMill [35], an automated system for running experiments on a distributed system, is part of the inspiration for NRG. However, DataMill does not provide the ability to conduct experiments on networks.

X. Conclusion

NRG enables profiling the performance of networked-applications in a reproducible manner. The toolset provides an insight into applications' performance, and can advise users and operators on better resource allocation and placement. NRG is an open source project, available at [5], [36]. This work raises no ethical concerns.

Acknowledgements

We thank Filip Ayazi for his contribution to the initial applications test framework. We thank Devang Sehgal for recent contributions to the development of NRG. We thank Stephen Ibanez for his contribution to the Debugging Performance Issues case study. We thank the anonymous reviewers and our Shepherd, Robin Marx, who helped us improve the quality of this paper. We acknowledge the support of Xilinx and Huawei. This work was partly funded by the Leverhulme Trust (ECF-2016-289), the Isaac Newton Trust, and UK's Engineering and Physical Sciences Research Council (EPSRC) under the EARL project (EP/P025374/1).

References

[1] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis *et al.*, "The case for RAMClouds: scalable high-performance storage entirely in DRAM," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 92–105, 2010.

[2] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the Killer Microseconds," *Commun. ACM*, vol. 60, no. 4, pp. 48–54, Mar. 2017.

[3] N. Zilberman, M. Grosvenor, D. A. Popescu, N. Manihatty-Bojan *et al.*, "Where Has My Time Gone?" in *PAM'17*, 2017.

[4] N. Zilberman, Y. Audzevich, G. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as Research Commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, September 2014.

[5] N. Zilberman, A. W. Moore, B. Cooper, J. Woodruff *et al.*, *NRG Repository*, https://github.com/NetFPGA/NRG-live.

[6] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz *et al.*, "Reproducible network experiments using container-based emulation," in *Proc CoNEXT*. ACM, 2012.

[7] L. Rizzo, "Dummynet: A Simple Approach to the Evaluation of Network Protocols," *SIGCOMM CCR*, vol. 27, no. 1, pp. 31–41, Jan. 1997.

[8] S. Hemminger, "NetEm - Network Emulator," http://man7.org/linux/man-pages/man8/tc-netem.8.html, [Online; accessed March 2021].

[9] L. Nussbaum and O. Richard, "A comparative study of network link emulators," in *SpringSim*, 2009, p. 85.

[10] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *IMC*. ACM, 2010, pp. 267–280.

[11] A. Roy, H. Zeng, J. Bagga, G. Porter *et al.*, "Inside the social network's (datacenter) network," in *ACM SIGCOMM CCR*, vol. 45, no. 4. ACM, 2015, pp. 123–137.

[12] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman *et al.*, "OSNT: open source network tester," *IEEE Network*, vol. 28, no. 5, pp. 6–12, 2014.

[13] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4→NetFPGA Workflow for Line-Rate Packet Processing," in *FPGA '19*. ACM, 2019, pp. 1–9.

[14] N. Sultana, S. Galea, D. Greaves, M. Wójcik *et al.*, "Emu: Rapid Prototyping of Networking Services," in *USENIX ATC*, 2017.

[20] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson *et al.*, "Scaling distributed machine learning with in-network aggregation," in *NSDI'21*, 2021, pp. 785–808.

[15] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *ACM SIGCOMM CCR*, vol. 41, no. 4. ACM, 2011, pp. 242–253.

[16] AWS, "Amazon EC2 FAQs - how many instances can i run in Amazon EC2?" https://aws.amazon.com/ec2/faqs/#how-many-instances-ec2, [Online; accessed March 2021].

[17] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang *et al.*, "Workload Analysis of a Large-scale Key-value Store," in *SIGMETRICS '12*, New York, NY, USA, 2012, pp. 53–64.

[18] D. Firestone, A. Putnam, S. Mundkur, D. Chiou *et al.*, "Azure accelerated networking: Smartnics in the public cloud," in *NSDI'18*, 2018, pp. 51–66.

[19] J. Woodruff, A. W. Moore, and N. Zilberman, "Measuring burstiness in data center applications," in *ACM Workshop on Buffer Sizing*, 2019, pp. 1–6.

[21] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang *et al.*, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS/PERFORMANCE*, 2012, pp. 53–64.

[22] C. Kim, A. Sivaraman, N. Katta, A. Bas *et al.*, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, 2015.

[23] D. Ardelean, A. Diwan, and C. Erdman, "Performance analysis of cloud applications," in *NSDI'18*, Apr. 2018, pp. 405–417.

[24] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira *et al.*, "Network Requirements for Resource Disaggregation," in *OSDI'16*, 2016, pp. 249–264.

[25] A. Faisal, D. Petriu, and M. Woodside, "Network Latency Impact on Performance of Software Deployed Across Multiple Clouds," in *CASCON '13*, 2013, pp. 216–229.

[26] S. Cheshire, "It's the Latency, Stupid," http://www.stuartcheshire.org/rants/Latency.html, may 1996, [Online; accessed March 2021].

[27] J. Dean and L. A. Barroso, "The Tail at Scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.

[28] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar *et al.*, "Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center," in *NSDI'12*, 2012.

[29] A. Ousterhout, J. Fried, J. Behrens, A. Belay *et al.*, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *NSDI'19*, 2019.

[30] G. Aceto, A. Botta, W. De Donato, and A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013.

[31] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal *et al.*, "Language-directed hardware design for network performance monitoring," in *SIGCOMM '17*. ACM, 2017, pp. 85–98.

[32] A. Gupta, R. Harrison, M. Canini, N. Feamster *et al.*, "Sonata: Query-driven streaming network telemetry," in *SIGCOMM'18*, 2018, pp. 357–371.

[33] Q. Huang, H. Sun, P. P. Lee, W. Bai *et al.*, "Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy," in *SIGCOMM'20*, 2020, pp. 404–421.

[34] D. Saucez, L. Iannone, and O. Bonaventure, "Evaluating the artifacts of SIGCOMM papers," *ACM SIGCOMM CCR*, vol. 49, 2019.

[35] J.-C. Petkovich, A. Oliveira, Y. Zhang, T. Reidemeister *et al.*, "DataMill: a distributed heterogeneous infrastructure for robust experimentation," *Software Practice and Experience*, vol. 46, p. 29, 2016.

[36] N. Zilberman, *NRG Webpage*, https://eng.ox.ac.uk/computing/projects/networked-systems/nrg/.