Evaluating Software Switches: Hard or Hopeless?

Vivian Fang
Tamás Lévai
Sangjin Han
Sylvia Ratnasamy
Barath Raghavan
Justine Sherry

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-136
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-136.html

October 12, 2018

Evaluating Software Switches: Hard or Hopeless?

Vivian Fang
UC Berkeley, Nefeli Networks

Tamás Lévai
BME

Sangjin Han
UC Berkeley, Nefeli Networks

Sylvia Ratnasamy
UC Berkeley, Nefeli Networks

Barath Raghavan
ICSI, Nefeli Networks

Justine Sherry
CMU, Nefeli Networks

ABSTRACT

Network virtualization, network functions virtualization, experimental switching and routing protocols, and many other applications rely on software switches. In the wake of these demands, industry and research efforts now develop a multitude of switches, *e.g.* Open vSwitch, FD.io VPP, BESS, Lagopus, Open Contrail, and Linux Kernel routing.

Given this proliferation of switch implementations, researchers face obvious questions for evaluation. Which switch is best? Or, more realistically, what workload features impact performance? What trade-offs are implied by different design choices in each switch implementation?

Our own efforts to answer such straightforward questions have nonetheless met roadblocks – for reasons that, to our knowledge, have never been explicitly studied or reported. In this paper, we report on the challenges in performing a fair and meaningful evaluation. We propose some strawman solutions to these challenges, but look to the community for constructive conversation regarding our challenges as well.

1 INTRODUCTION

Today's software switch ecosystem offers countless options for deployments servicing widespread use-cases; from network functions virtualization (NFV) to data center multitenancy and network virtualization, to experimentation and new protocol designs. The combined market for SDN, NFV, and Virtual Networking all of which integrally involve software switching is projected to reach $18 billion by 2020 [28]. Faced with so many choices of software switches – VPP [15], Open vSwitch (OVS) [27] and OVS-DPDK [2], BESS [17], Open Contrail [1], ESwitch [24], Lagopus [30], Linux kernel routing [10] and others – a network engineer will naturally be faced with deciding which switch is best for their needs.

Understanding which switch is "best", however, entails analysis of numerous factors: what performance metrics matter? What workload factors impact each metric? Further, for researchers and developers seeking to design the next generation of improved software switches, we might also ask: are there fundamental tradeoffs in the design choices that different systems make? What implementation choices in switches make the most impact on latency, throughput, or jitter?

To date, no comprehensive evaluation of these questions exists. Publicly distributed industry reports focus on narrow test cases, and are often designed to demonstrate that a particular switch "wins", with insufficient information to reproduce results [14, 33]. Many of these reports are designed more as marketing material than honest evaluation. More academic evaluations typically focus on the design and implementation of the authors' novel switch design and hence evaluate their own system compared to a small selection (typically, only one) of other switches – with narrow experiments designed to focus on the core features of the new switch. For example, the NSDI OVS paper primarily focuses on the performance and benefits of its novel flow-caching strategies, but offers little in the way of head-to-head comparisons against its competition [27]. The PISCES paper similarly focuses on its compiler and ability to re-use the OVS architecture for P4 programs – but does no comparisons against other programmable switches [34]. A few academic efforts do perform third-party evaluations of switches, but are limited in scope – focusing on a single switch at a time [5, 13], or a few point design comparisons between switches [19].

In short, software switches are a crucial component of network infrastructure today – and with so many switches on the market, we are in the midst of a battle both for market share and for what technical designs [26] will win out. Nonetheless, efforts to date to evaluate these switches head-to-head fall short of providing clarity in this battle.

We too fell short of evaluating software switches!

We set out to do a systematic, comprehensive, and unbiased comparison of software switches. Yet, we were repeatedly stymied, often for reasons that to our knowledge have never been explicitly reported or tackled.

In this paper, we instead report on why we stumbled, describing our experiences, providing some suggestions for progress, and proposing some open questions that remain in our way. That is, our goal is not to identify the "winner" or answer questions regarding the ideal software switch design (we leave this to future work). Instead, we ask: is such an evaluation even feasible? What can we do to make such an evaluation both tractable and meaningful?

In what follows, we identify six challenges that face a fair and comprehensive evaluation; for each challenge we either propose a strawman solution or leave open the question(s) of how to evaluate the challenge.

- Testing systems requires configuring them for peak performance on the hardware they're deployed on. How do we do this?
- Performance varies across so many dimensions – *e.g.* packet size, flow duration, number of concurrent flows. How do we measure which switches have the most performance "stability" despite fluctuations across all dimensions?
- Does switch architecture matter more than implementation?
- How can we quickly identify which algorithmic/design differences in switches impact performance the most – without reading thousands of lines of code?
- How do we standardize what metrics switch users – like datacenter and telco operators – care most about?

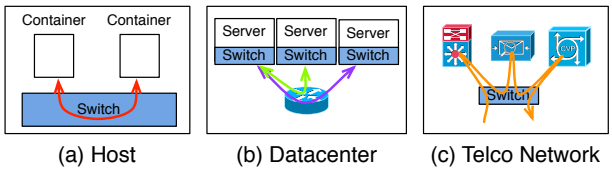

Figure 1: Three usage scenarios for software switches.

- How do we evaluate seemingly unquantifiable features (*e.g.* feature completeness or external tool compatibility) that do not relate to system performance?

We ask the community for their proposals and solutions as well, not just for the benefit of our own research, but so that we as a community can arrive at shared conclusions for the best way to evaluate software switches (and perhaps even generalize our shared conclusions to other networked systems as well).

2 BACKGROUND

Software switches are far richer and more extensible than standard L2 'switching'. In Figure 1, we illustrate three common deployments for software switches. In (a), we see a switch steering traffic between co-resident containers (the same approach can be used to steer traffic between VMs and processes); the switch may also implement some access control policies between containers. In (b), we see switches deployed on hosts within a datacenter; here the switches are used in instantiating separate overlay networks over a shared datacenter fabric (implementing virtual networking). The switches may add additional (VXLAN, Ethernet, even IP) headers, drop packets which violate policy, and steer packets within the local host. In (c), we see a telecommunications network using a software switch for routing in NFV, steering packets between a sequence of network functions (NFs); this type of steering can be done by inspecting and modifying a Network Services Header (NSH). Not shown, software switches can also be used for research and experimentation, as a fast way to prototype experimental protocols.

All switches have some number of *ports* – which may be physical (connected to a NIC) or virtual (connected to a container or VM). Connecting these ports is a switch's software fabric, the code which decides where to steer packets, when to drop packets, or how to modify packets. This fabric may be implemented in the kernel (*e.g.* Linux Bridging and OVS) or in user space (*e.g.* OVS-DPDK, Lagopus, VPP, or BESS). Software switches usually involve multiple cores carrying packets from the input port, through the fabric, and to the output port in parallel.

The switch fabric usually consists of a sequence of discrete modules; these modules may consist of arbitrary C++ code (a design approach proposed by Click [21]) or they may consist of match-action tables (as popularized by OVS). The lead developer of OVS refers to switches with C++ modules as 'code driven' switches, and switches with table-based modules as 'data driven' switches [26]. However they are implemented, switch code is typically tightly tuned for performance, aiming for latencies in the 10s of μs and throughputs in the 10s of millions of packets per second (Mpps).

3 CHALLENGES

We now discuss our six challenges for software switch evaluation. In our discussions, we provide experimental data from our own testbed. For all provided data, we use the following setup with two identical servers. One server serves as the packet generator and the other serves as the device under test (DUT) running the software switch and (when relevant) VMs. Each server has two 14-core Intel Xeon E5-2690 v4 2.6GHz CPUs that have HyperThreading disabled, and two single-port Intel XL710QDA2 40Gbps NICs.

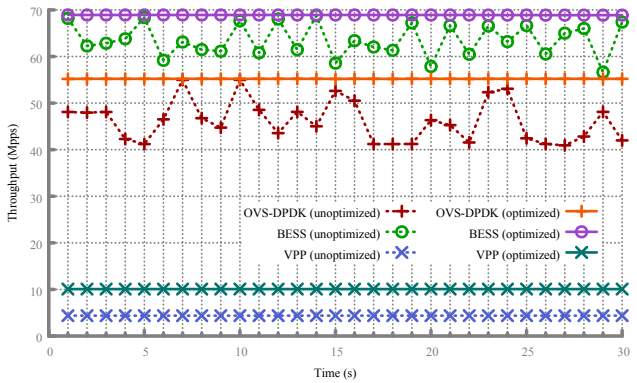

Figure 2: BESS, OVS-DPDK, and VPP throughput with their respective tuned GRUB configuration and with the default Ubuntu GRUB configuration. Correctly tuning GRUB configuration leads to improvements in throughput, as well as reductions in throughput variance.

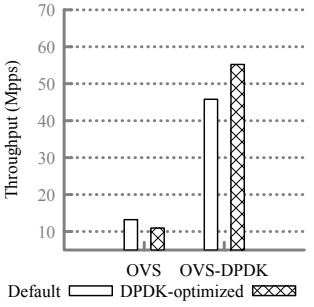

Figure 3: Throughput for OVS and OVS-DPDK with/without a DPDK-optimized GRUB

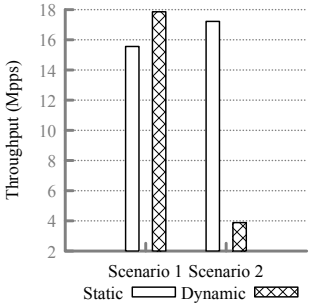

Figure 4: Throughput for two BESS pipelines using either the dynamic or static scheduler

3.1 Challenge #1: System Configuration

Researchers usually engineer experimental testbeds for software switches to provide the latest hardware features, ensuring that, *e.g.*, the testbed has enough memory and NIC capacity for high throughput. However, for all switches we have worked with, READMEs and installation guides only provide the minimum instructions to simply *get the switch running*. In our experience, there are many kernel parameters, BIOS configurations, operating system settings, and architecture-specific GCC flags that must be modified to reach peak performance on the provided hardware. For example, it is standard practice to disable HyperThreading and energy savings features (both of which reduce throughput) in the BIOS; it is also very common to manually isolate and assign processor cores to OS processes, and to use different compiler parameters. None of these practices are documented in any switch README or SETUP files, but are rather recorded by third-parties [9] or have to be inferred from the switch's underlying implementation *e.g.*whether it is built

on DPDK, how it schedules processes, *etc.*To perform a fair comparison, researchers need to evaluate each switch under its own peak configuration/installation.

Tuning system parameters improves performance significantly over default settings. In Figure 2, we show throughput for BESS [17], OVS-DPDK [2], and VPP [15] running an empty processing pipeline, each with a standard GRUB (Linux boot loader) configuration and an optimized GRUB configuration. Some examples of changed settings in each configuration include: disabling energy-saving settings and HyperThreading in the BIOS, setting larger hugepages, and isolating switch-dedicated of CPUs from the kernel scheduler (note that this does not include parameters set in the OS, such as core-to-process allocation or scheduler selection and customization). For all three switches, using the standard GRUB configuration leads to lower throughput – and for OVS-DPDK and BESS, the default GRUB also leads to significant throughput instability. VPP has consistent throughput in both cases because it automatically pins cores; and the increase in throughput is also attributed to specifying which cores to pin.

Because the GRUB configuration parameters are detailed, have a significant impact on throughput, but are unspecified, their existence makes it hard for parties not closely involved with switch development to be sure they have fairly measured a switch's performance results.

Parameter tuning for switches is not one-size-fits all: different switches require different settings. Making things even more challenging for evaluators, we cannot simply specify a fixed set of system configuration parameters for all switches, as different switches require different settings. In Figure 3, we plot throughput for OVS-DPDK [2] and regular OVS [27] using each of two GRUB configurations: one configuration which isolates cores from the OS kernel and one which does not. User-space applications like OVS-DPDK benefit from core isolation because it prevents the kernel from assigning additional processes to the same core, competing for resources with the switch. However, where OVS-DPDK benefits from this optimization, base OVS actually suffers a 17.3% decrease because OVS carries packets through the kernel, and consequently has fewer cores to schedule packet processing tasks.

Even the same switch may require different settings given different workloads and use-cases. Given the previous two examples, one might be prepared to package one set of configuration parameters for each switch. Unfortunately, one cannot package a set of parameters and expect it to perform ideally, even in the context of a single switch as different workloads and use cases can require different settings. For example, BESS [17] offers two different schedulers: a dynamic, pushback-based [22] scheduler which detects forwarding pipeline bottlenecks and adjusts task service times to avoid wasted work, and a default scheduler which does not detect pipeline bottlenecks and serves all forwarding tasks according to a static policy. Figure 4 shows throughput with the dynamic and static scheduler for two switch workloads:

- Scenario 1 has one core serving two packet processing tasks, one which forwards packets to a slow NF, and one which forwards packets to a fast NF. The dynamic scheduler detects that one task is servicing fewer packets and thus allocates more CPU time to the faster worker, increasing net throughput relative to the default.
- Scenario 2 has one core serving one worker, but this worker splits incoming packets between delivering them to a slow NF and a fast NF. The dynamic scheduler observes that not all packets are being served (because of the slower NF) and hence schedules less CPU time to the task – leading to fewer packets being delivered to the fast NF and a lower aggregate throughput than the default.

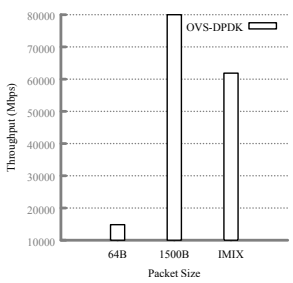

Figure 5: Throughput for OVS-DPDK for various packet sizes.

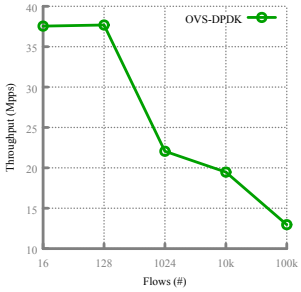

Figure 6: Throughput for OVS-DPDK for an increasing number of flows.

OPEN QUESTION: To guarantee fair comparison, how can evaluations of software switches evaluate each given switch and workload under ideal configuration parameters?

STRAWMAN: Switches should be able to "self-tune" their configurations on installation. Designing switches to do so is, however, non-trivial – a switch needs to look at a variety of hardware options and with little or no knowledge about the incoming workload make all the right choices about scheduling, BIOS options, kernel parameters, etc. Hence, we believe this is an open research question. It is also worth noting that self-tuning switches would not only make evaluation easier, but make deployment of software switches easier for network administrators!

3.2 Challenge #2: Performance Stability

Once an evaluator has configured their testbed (as discussed in the previous section), they must then consider the traffic workloads over which to test the switch. These test parameters include offered load, the number of concurrent flows, the flow arrival rate, distribution of packet sizes, flow duration distribution *etc.* In live production deployments, all of these parameters can change dynamically *e.g.*, as usage changes with time of day or as flash crowding leads to bursty, unexpected changes in traffic characteristics. Hence it is no surprise that in our conversations with operators, we have learned that they value *performance stability* – that the switch will provide good throughput, latency, and jitter under a range of these traffic parameter configurations.

Nonetheless, in research evaluations there is a tendency to show isolated data points as only a single parameters. For example, Figure 5 shows the median throughput as the packet size changes (64B, 1500B, and simple IMIX [4]) for OVS-DPDK configured with 10,000 IP-based forwarding entries. For this test, the traffic load includes 1024 concurrent flows with a flow arrival rate of 5 seconds. Figure 6 shows the average throughput for the switch configuration and

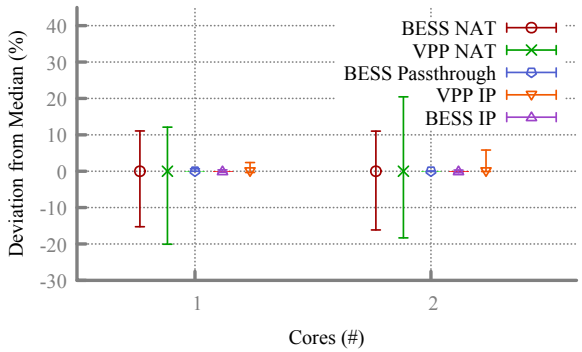

Figure 7: BESS and VPP NAT and IP routing, plotted with parameter error bars.

same traffic load, but we vary the number of active flows and fix packet size at 64B.

Both of these figures show that OVS-DPDK is sensitive to both packet size and flow arrival rate – but what they do not illustrate is the *range* of throughputs an operator can expect from OVS-DPDK as these parameters and others all change simultaneously.

OPEN QUESTION: How do we quantify or visualize performance stability – the range of switch behavior given variations in workload?

STRAWMAN: We suggest a simple visualization called parameter error-bars. Traditionally error bars are used to show statistical variation for a fixed workload – we suggest using error bars to show variation over the entire workload parameter space. Consider a traffic generator with options to adjust packet size, flow duration distribution, flow arrival rate, and average number of concurrent flows. An evaluator might run experiments by sampling from the Cartesian product of *all* parameters, and then plot the median result along with error bars showing the maximum and minimum results across all runs. Figure 7 shows a comparison between VPP and BESS both running a NAT and a L3 Forwarder, now with a sampling of changes in all variables. The peak of the error bar shows the maximum observed throughput, and the bottom shows the minimum, both normalized relative to the median observed throughput. As a baseline, we plot parameter error bars for BESS passthrough forwarding (which performs no operations over packets) as well.

In this case, we see that the NAT implementations of BESS and VPP are much more variable than passthrough routing or IP routing as the workload for each implementation changes. BESS's IP routing implementation is more stable than VPP's. One interpretation is that VPP (which has 3-4× less throughput than BESS) is more variable because there are more stages in processing a packet, and each stage/step has some amount of variability which compounds. Most importantly, for this discussion, stability is a feature we would not have observed without plotting our results in this fashion.

3.3 Challenge #3: Does Architecture Matter?

As researchers, we often focus on system architectures and seek to explain performance differences as stemming from fundamental architectural differences. However, this tendency is not limited to researchers. In an influential keynote presentation and blog post, Ben Pfaff, the lead developer of OVS, characterized software switches as following one of two architectural designs: 'code-driven' or 'data-driven.' This characterization has been widely embraced in industry and results comparing, *e.g.* OVS and VPP [3], are viewed as shedding light on which architectural choice wins.

While perhaps an unpopular position, in our experience, architectural differences of the form Pfaff raises are typically a secondary factor at best in explaining performance differences. Instead, it is more common that algorithms and implementation techniques – applicable to any architecture (with some non-trivial developer effort) – bring the majority of performance gains.

For example, OVS by default can forward only 1.68 Mpps using a single core, with latencies averaging 129.65μs – an order of magnitude worse than switches like Lagopus, VPP, and BESS. The primary source of this performance gap between these switches is not architectural, but that OVS runs through the kernel while the latter three switches are built with kernel bypass through DPDK and/or SR-IOV [12]. Changing this one implementation choice brings OVS-DPDK up to 12.92Mpps and 18.2μs of latency, making its performance comparable to competitors (both code- and data-driven).

To name a few more examples, tuple-space search algorithms for table lookups [35], Read-Copy-Update rather than traditional locking [23], and packet batching [20] have all brought substantial benefits to switching in both code- and data-driven designs (and these benefits far outweighing any apparent differences from the architecture itself).

OPEN QUESTION: Is our claim above – that architectural differences are secondary when it comes to performance – true? Our hypothesis is based on point observations; how can we more rigorously show this?

3.4 Challenge #4: Identifying and Isolating Implementation Differences

It is common for benchmarks to evaluate different switches for particular tasks, *e.g.*, 'L3 Forwarding' or 'NAT'. Internally, however, these may be implemented using different algorithms – and these differences between algorithms lead to big differences in performance.

For example, a 2015 study evaluated throughput sensitivity to concurrent flows, comparing VPP and OVS-DPDK on basic L2/L3 routing workloads [3], concluding that VPP maintained higher throughputs (up to 38× faster) than OVS-DPDK as the number of concurrent flows increased. However, a deeper inspection of the match/lookup algorithms in each reveals a more complex story. OVS-DPDK uses a novel flow-caching strategy [27] to reduce lookup times for complex multi-table lookups required for network virtualization. This flow-caching strategy introduces an additional step in packet processing that increases per-flow overheads for short, single-table lookups (*e.g.* L2/L3 switching), but the authors of OVS claim it improves throughputs for longer, multi-table tasks (*e.g.* network virtualization). Hence, 'bake-off' style evaluations like the 2015 study fall short in two ways: (a) identifying that there *are* algorithmic differences and (b) benchmarking the differences in performance between the algorithms in isolation of other design choices and optimizations within the switch implementations and showing their strengths (*e.g.*, multi-table lookups) and weaknesses (*e.g.*, large numbers of independent flows).

To do so, one can always detect algorithmic differences via a detailed reading of the code, but this is a painstaking and slow process (e.g., BESS has 23k lines of C++ code, OVS has 255k lines of C code, and VPP has 312k lines of C code). Expecting an individual developer to become intimately familiar with the codebases for even just two switches is unreasonable. Furthermore, analyzing the impact of the algorithm requires both an analysis of the algorithm's *complexity* and *empirical performance* (given, *e.g.*, its amenity to cache-tuning or optimizations for mice vs. elephant flows). This empirical testing must be considered in isolation of the existing performance optimizations in the switch (that is, independently of the rest of the switch codebase).

OPEN QUESTION: Can we automate or aid the process of analyzing what algorithmic differences exist between switches, and to what extent these differences impact various performance properties?

STRAWMAN: A growing body of research in the programming languages community shows how to automatically extrapolate performance characteristics (memory utilization, expected CPU usage) from raw source code and a hardware specification [18]. Indeed, some such tools are now practical – the Google Benchmark [16] library can estimate an algorithm's big-O when it is microbenchmarked. More recent work has moved in to the automatic comparison of different switches [8]; can such work be extended to diagnosing performance differences in switches?

3.5 Challenge #5: Unoptimized Metrics

Performance studies typically focus on switch throughput and latency – yet there are many other performance figures of merit for telcos, datacenters, and other users. For example, jitter, loss rate, and rule-update-time are far less frequently measured. The problem that results is that metrics that aren't measured can't be optimized; nor can switches compete over better performance with regard to unmeasured metrics.

One reason jitter and loss are somewhat ignored is that it is difficult, if not impossible, to match the statistics reported by hardware switches, which can guarantee consistent latencies (introducing little to no jitter beyond that introduced by queueing) and can guarantee no packet loss (once again, except for that introduced by queueing). Software switches struggle to make such strong guarantees because OS interrupts, multicore contention, and even branching and varied compiler output lead to unpredictable timing for packet processing – furthermore, software switches are often pushed to implement more features than typically expected of a hardware switch (*e.g.*, NATing or adding virtual network headers).

The developers of BESS describe [25] one experience where a large telco measured BESS for loss-free forwarding (following guidelines in RFC 2544 [6]) and found the estimated loss-free forwarding rate continuously decreased due to unexplained packet losses lasting for tens or even hundreds of milliseconds. Modifying the queue length did not eliminate the losses, nor did changing core isolation, ACPI settings, turning off HyperThreading and turbo boost, or eliminating all other processes from the machine. Upon further investigation, they found the culprit: transparent hugepage support in Linux, which would periodically be scheduled on a BESS worker core, causing the queue to overflow. Disabling this feature reduced the (non-overflow derived) losses to zero.

While the fix required no new algorithms or design choices, it took significant developer time to resolve – and no one had paid attention to the problem prior to the telco's own measurements.

OPEN QUESTION: How can we allow switch users to standardize what features matter in software switches so that switch developers can optimize for them?

STRAWMAN: It may be time for standardized software switch benchmarks. Many other domains have benefited from standardized benchmarks, *e.g.*, Acid testing [36] for web browsers, and machine learning datasets for classifiers [7, 32]. There has been work in benchmarking OpenFlow controllers [31], and switches that use P4 [11]. Benchmarks have many benefits beyond allowing switch users to direct developer attention. Having switches run the same set of benchmarks can prevent industrial evaluations from cherry-picking or custom-designing experiments for marketing purposes: standard workloads mean everyone runs the same experiments. Benchmarks also allow 'bake-off' style evaluations to direct performance tuning so that researchers can focus instead on algorithmic and design questions driving each metric's results.

3.6 Challenge #6: Performance isn't everything

In the previous three sections, we discussed challenges impacting performance measurements (primarily of throughput). In practice, network operators who deploy software switches also consider many harder-to-quantify attributes such as feature completeness, compatibility with external tools (like SDN controllers or monitoring/statistics reporting suites), system flexibility, hardware compatibility, and ease of programming. Yet these harder-to-measure attributes – such as feature completeness and compatibility with external tools – are deployment obstacles.

Some of these attributes may be rightfully dismissed as simply engineering time – and not needful of research time. For example, feature completeness is often just a checklist of developer tasks to complete. Most switches do not yet support NSH [29], but adding this capability is merely a few hours of developer time in BESS, OVS, or VPP. Nonetheless, we believe that *system flexibility* and *ease of programming* stem from the system architecture (returning to Pfaff's taxonomy [26]).

System Flexibility: Features like NSH may be easily grafted onto any switch architecture. However, consider the following two extensions:

- URL-based routing (*i.e.*, L7 load balancing) does not appear to be easy to integrate onto OVS/OVS-DPDK or Lagopus. These are largely stateless switches, but URL-based routing requires stateful flow reconstruction and TCP termination. This extension can be implemented in VPP and BESS due to their code-driven, stateful designs.
- Flow caching for table lookups, which is an important performance optimization in OVS/OVS-DPDK and Lagopus, would be difficult to graft onto BESS without introducing monolithic and universal code, breaking the BESS design pattern of small, independent module implementations.

Hence the lack of these two features is not a mere 'todo' for developers, but expose questions about the system architecture altogether and whether or not the switch can or should support them at all.

OPEN QUESTION: Can we quantify or systematize system flexibility? After conversations with software engineering researchers, we believe this to be an open problem.

Ease of Programming: VPP, BESS, and OVS make dramatically different choices about the API they expose to operators. OVS expresses rules through match-action table entries. Although BESS and VPP are both code-driven switches, BESS exposes an interface for constructing pipelines using available modules, while VPP provides a command-line interface for configuring individual modules. We have yet to see research discussing which API is best. The status queue for evaluating ease of programming in systems papers seems to be measuring the number of lines of code in a switch configuration program, despite the fact that 'one-liner' programs are often considered a mark of clever, non-intuitive thought!

OPEN QUESTION: Can we quantify or systematize ease of switch programming?

4 DISCUSSION

This concludes our presentation of our evaluation challenges. We remain unprepared to answer our titular claim (Is a comprehensive, unbiased evaluation 'hard or hopeless?') Nonetheless, our attempts to understand why our evaluations have failed have led us to realize that how to design a strong evaluation is as much of a research question as whether or not any given switch architecture wins out.

In addition to our 'open questions' above, we conclude upon two meta-questions. First, which of these challenges apply to other system evaluations? For example, one can imagine that identifying and isolating implementation differences (§3.4) is a challenge that may also impact databases or analytics frameworks. Second, which challenges don't seem to apply to some other systems and why? For example, many distributed systems papers often appear to provide analytical results that are independent of engineering challenges such as system tuning, §3.1, instead focusing purely on algorithms. Why should software switches be any different than other software systems?

REFERENCES

[1] Open Contrail. http://www.opencontrail.org/.

[2] Open vSwitch with DPDK. https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview.

[3] Validating Cisco's NFV Infrastructure Pt. 1. http://www.lightreading.com/nfv/nfv-tests-and-trials/validating-ciscos-nfv-infrastructure-pt-1/d/d-id/718684?page_number=8.

[4] Wikipedia: Internet mix. https://en.wikipedia.org/wiki/Internet_Mix.

[5] A. Bianco, R. Birke, L. Giraudo, and M. Palacin. OpenFlow Switching: Data Plane Performance. In *2010 IEEE International Conference on Communications*, pages 1–5, May 2010.

[6] S. Bradner and J. McQuaid. Benchmarking Methodology for Network Interconnect Devices. RFC 2455.

[7] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[8] E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 316–329, New York, NY, USA, 2017. ACM.

[9] E. Chaudron. Measuring and comparing Open vSwitch performance. https://developers.redhat.com/blog/2017/06/05/measuring-and-comparing-open-vswitch-performance/, June 2017.

[10] V. Danen. Understanding the basics of Linux routing. http://www.techrepublic.com/article/understand-the-basics-of-linux-routing/, 2001.

[11] H. T. Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon. Whippersnapper: A p4 language benchmark suite. In *Proceedings of the Symposium on SDN Research*, pages 95–101. ACM, 2017.

[12] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

[13] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle. Performance characteristics of virtual switching. In *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, pages 120–125, Oct 2014.

[14] FD.io. VPP: Performance Expectations. https://wiki.fd.io/view/VPP/What_is_VPP%3F#Performance_Expectations.

[15] FD.io VPP. https://wiki.fd.io/view/VPP.

[16] Google. Google benchmark. https://github.com/google/benchmark.

[17] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. Softnic: A software nic to augment hardware. *Dept. EECS, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-155*, 2015.

[18] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *38th Symp. on Principles of Prog. Langs. (POPL'11)*, pages 357–370, 2011.

[19] R. Kawashima, S. Muramatsu, H. Nakayama, T. Hayashi, and H. Matsuo. A Host-Based Performance Comparison of 40G NFV Environments Focusing on Packet Processing Architectures and Virtual Switches. In *2016 Fifth European Workshop on Software-Defined Networks (EWSDN)*, pages 19–24, Oct 2016.

[20] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon. The power of batching in the click modular router. In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, pages 14:1–14:6, New York, NY, USA, 2012. ACM.

[21] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[22] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 71–84. ACM, 2017.

[23] P. E. McKenney and J. D. Slingwine. Read-copy-update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Oct. 1998.

[24] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor, F. Juhász, A. Kőrösi, and G. Rétvári. Dataplane Specialization for High-performance OpenFlow Software Switching. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 539–552. ACM, 2016.

[25] Nefeli Networks. Private commn., 2017.

[26] B. Pfaff. Converging Approaches in Software Switches. Keynote Speech, SIGOPS Asia-Pacific Workshop on Systems (APSys), June 2016.

[27] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, 2015. USENIX Association.

[28] PR Newswire. The SDN, NFV & Network Virtualization Ecosystem: 2016 - 2030. http://prn.to/1Tw0dJO.

[29] P. Quinn, U. Elzur, and C. Pignatero. Network Services Header. IETF Draft draft-ietf-sfc-nsh-18, July 2017.

[30] R. Rahimi, M. Veeraraghavan, Y. Nakajima, H. Takahashi, S. Okamoto, and N. Yamanaka. A high-performance openflow software switch. In *High Performance Switching and Routing (HPSR), 2016 IEEE 17th International Conference on*, pages 93–99. IEEE, 2016.

[31] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, A. W. Moore, et al. Oflops: An open framework for openflow switch evaluation. In *PAM*, volume 7192, pages 85–95. Springer, 2012.

[32] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[33] J. Scheurich and M. Gray. OvS-DPDK performance optimizations to meet Telco needs. Open vSwitch Fall 2016 Conference.

[34] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 525–538. ACM, 2016.

[35] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 135–146, New York, NY, USA, 1999. ACM.

[36] Web Standards Project. Acid Tests for Web Browsers. http://www.acidtests.org/.