Thoughts on Load Distribution and the Role of Programmable Switches

James McCauley
UC Berkeley and ICSI
jmccauley@eecs.berkeley.edu

Aurojit Panda
New York University
apanda@cs.nyu.edu

Arvind Krishnamurthy
University of Washington
arvind@cs.washington.edu

Scott Shenker
UC Berkeley and ICSI
shenker@icsi.berkeley.edu

ABSTRACT

The trend towards powerfully programmable network switching hardware has led to much discussion of the exciting new ways in which it *can* be used. In this paper, we take a step back, and examine how it *should* be used.

CCS CONCEPTS

• **Networks** → *Programming interfaces*; *Network management*;

KEYWORDS

Architecture, Programmable Hardware

1 INTRODUCTION

Fastpath forwarding in hardware switches has long been implemented using switching ASICs. Traditionally, switch ASICs have been fixed function and are designed to forward packets sent using a small set of network protocols (*e.g.,* IP packets sent over Ethernet). The forwarding behavior of these ASICs – *i.e.,* how they process packets – is governed by one or more forwarding tables that contain forwarding rules, each of which is expressed as a match-action pair. The match portion of the rule acts as a predicate to select whether a rule applies to a packet, and is expressed in terms of protocol-specific header fields. The action specifies how a matched packet should be forwarded and can both specify header modification (*e.g.,* reducing the TTL of an IP packet) and forwarding instructions (*e.g.,* specifying that a packet should be sent out through port X). While the use of fixed function ASICs enabled switches to forward terabits of aggregate traffic every second, their inflexibility has been an impediment in adopting new protocols. As a result, handling new protocols at high speed has typically required a hardware upgrade and a new generation ASIC.

In response to this limitation, switch vendors developed programmable switch ASICs, *e.g.,* Barefoot's Tofino [3], Cavium's Xpliant [5], Intel's Flexpipe [19], Cisco's Doppler [18], Broadcom's Trident 3 [7]. Several of these ASICs implement the Protocol Independent Switch Architecture (PISA), and all of them enable a much greater degree of protocol independence (*i.e.,* the ability to programmatically add support for new protocols to existing switch hardware). Protocol independence in these ASICs is enabled by two main features:

- Flexible packet parsing: this allows a programmer to specify how a packet should be parsed and the set of header fields that should be extracted (these fields can then be used when matching packets and modified by forwarding actions).
- Compound actions: these allow programmers to compose multiple simple switch actions into a more complex forwarding action that can be applied to packets.

Network programmers can specify the forwarding behavior of these switches by providing a description of how packets should be parsed, a set of compound actions, and forwarding rules. The move to programmable switches can greatly improve network flexibility, allowing operators to easily adopt new network protocols. As a result, many switch vendors, such as Cisco [6] and Arista [2], have announced switches built on these programmable platforms.

While the primary motivation behind programmable switches is to enable the adoption of new network protocols, a goal that few could argue with, the flexibility offered by programmable switches has also been recently used to implement application logic in switches. Some of the recent proposals that have looked at implementing application functionality in switches have addressed consensus protocols (NetPaxos [8], NetChain [22]), caching for key-value stores (NetCache [23]), load balancing for web services and other applications (SilkRoad [25], Packet Subscriptions [21]), aggregation for distributed computation (DAIET [32]), and stream processing (Linear Road [20]). These proposals rely on both flexible packet processing and compound actions to implement functionality that extends far beyond traditional packet forwarding.

In this paper, we consider the question of whether (and, if so, *which*) application functionality should migrate to switches. Note that this is not a question that can be answered by simply appealing to the end-to-end principle [31]. The end-to-end principle involves asking what functions belong in the network abstraction (*i.e.,* should reliability or multicast be part of the network layer?), whereas the proposals under consideration here are merely moving some application functionality out of servers and into switches that are both located in the same managed infrastructure (typically a datacenter). Thus, this question is not about network abstractions, but whether functionality should be implemented in switch hardware or server hardware in a particular deployment.

We start by looking at how this question plays out in load balancing (Section 2), and then consider the problem of load partitioning (Section 3) – two contexts where there have been proposals for moving application functionality into switches. We then summarize the lessons learned from these two examples in Section 4, and discuss the use of programmable switches for networking applications in Section 5, before concluding in Section 6.

2 LOAD BALANCING

2.1 Background

Load balancers are widely deployed as a mechanism to scale web services by spreading requests across multiple backend servers. Initially, load balancers were implemented as hardware middleboxes which offered a great deal of flexibility in how connections were mapped to backend servers. For example, F5's BIG-IP [13] hardware load balancer supports load balancing policies which can be based on application, resource utilization at a server, connection source, etc. While these load balancers offered great policy flexibility, they were not designed to scale. In particular, many policies require that state be synchronized across instances, which makes it challenging to add additional instances. One response to this has been to distribute policy enforcement, as is done in 6LB [9] via IPv6 Segment Routing [15]. While this approach avoids synchronization, the performance challenges remain the same. In particular, 6LB is implemented by forwarding each request through a sequence of load balanced servers until one of the servers deems it appropriate to respond. Thus, in the worst case for this approach, the initial packet for a request visits all equivalent servers (the average case behavior depends on the policy used, but for any performance-dependent policy, a request is likely to be received by more than one server). As a result, this approach adds additional load at each server for any request processed by the system, and also inflates response latencies. Ultimately, while this type of approach alleviates some of the concerns with traditional middlebox-based load balancers when applied to domains such as video streaming where response sizes are far larger than request sizes, it is not as appropriate for traditional web request handling where requests and responses are of roughly similar size and fit within one or a few packets.

A widely-employed response to this scaling challenge was a migration to scale-out server-based load balancers such as Maglev [12] and Ananta [30] that rely on consistent hashing to map each new connection to a backend server. The use of consistent hashing means that assigning connections to backend servers is stateless, so no cross-instance synchronization is needed on the initial assignment. While this choice enabled scaling, it had three main drawbacks.

First, it eliminated any policy flexibility (aside from which hash function was used and what header fields were hashed). While this made it hard to respect the flow affinities needed for some applications (*e.g.,* ftp), the larger issue is that random assignment of flows typically leads to load imbalances. If we assume all flows impose the same load, then the standard result for the random balls-and-bins model states the maximum load is $\Theta(\frac{\ln n}{\ln \ln n})$ with high probability. Since one cannot predict which server will have the overload, one needs to overprovision the total number of servers by this factor, which for large n can be significant.

Second, while the use of consistent hashing allows the initial processing of a connection to be stateless, the load balancers themselves must remain stateful. This is because correct load balancers need to maintain *per-connection consistency* – *i.e.,* they need to ensure that all packets in a connection are forwarded to the same backend server – despite changes in the set of backend servers (due to server failure or addition of new servers) or other events. Load balancers implement per-connection consistency by recording the backend server chosen to process the first packet in a connection (*e.g.,* a TCP SYN packet) in a *connection table* and then forwarding subsequent packets based on the information stored in the connection table.

Third, the fact that the load balancing was implemented in software led to limited performance. For instance, an optimized implementation of Maglev can handle up to 9 million packets per second per core with performance scaling sublinearly as we increase the number of cores [29]. As a result, operators commonly deploy several load balancers to handle the load for a single service, with an ECMP based switch or hardware load balancer responsible for partitioning connections across the main pool of load balancers. When deployed in this manner, connection tables can also be synchronized across load balancers in order to provide fault tolerance.

Recently SilkRoad [25] has proposed using programmable switches to implement high-throughput load balancers, thus reducing the number of instances required. In SilkRoad, the connection table is stored in SRAM connected to the switch ASIC and is updated whenever a new connection is assigned a backed server. While SilkRoad instances can process higher packet rates when compared to software load balancers, they can only handle a limited number of active connections. This is because switches generally have very limited SRAM available, *e.g.,* many current switches provide about 12MB [33] of SRAM per port. Once SRAM is exhausted, SilkRoad must either evict entries from the connection table – thus violating per-flow consistency – or rely on the control plane to manage the connection table – thus severely degrading performance. As a result, storing the connection table on switches severely limits the scalability of SilkRoad and similar approaches.

Thus, the move to switch hardware may have eliminated the concern about performance, but not about state or policy flexibility. In fact, the state limitations for switch implementations of load balancing is much more challenging than for server-based approaches. We now turn to how one can address the state and policy issues, without being limited by switch hardware.

2.2 Handling More Connections

We observe that when processing a packet, SilkRoad needs access to only the state that is associated with that connection. Thus, the information needed to process a single packet is of fixed size, and hence could potentially be encoded in the packet. This scheme was originally advocated by [27], which proposed encoding connection tracking information in the TCP timestamp option [4]. This information can also be embedded in other fields, including the QUIC connection ID [24] field, the MPTCP [16] destination port, or other protocol fields. When using MPTCP, another alternative is to use the mechanisms recently proposed as a part of the RFC6824BIS draft [10, 17], which are designed to enable static load balancing policies (*i.e.,* ones which hash the TCP 5-tuple) for MPTCP connections. This mechanism allows a server to return a new destination address (based on what server is chosen by the load balancer) when the client initiates the first subflow, and all subsequent subflows use this new destination address. In our case, we can therefore use all or part of this address to encode connection information, which is especially promising with large IPv6 addresses.

The use of mechanisms where load balancing information is encoded in the packet header requires two things: (i) client support – since the client must either enable TCP timestamps or it must use either QUIC or MPTCP, and (ii) parsing flexibility at the switch so it can extract and interpret information in the packet header to determine the appropriate backend server (that is, the switch must be programmed to make a forwarding decision based on a packet header not typically used for forwarding decisions). However, assuming the client supports this mechanism (*i.e.,* implements and enables support for TCP timestamps or uses QUIC or MPTCP), load balancing in this manner achieves the same performance as can be achieved using SilkRoad without imposing a limit on the number of connections that can be handled by a load balancer.

Furthermore, even when client support is absent, load balancing can be implemented without putting connection tracking information in switches by storing this information in the backend servers using a mechanism proposed previously in Beamer [28]. In this case, we rely on the observation that the only case where consistent hashing will fail to forward the packet to the correct backend server is when the backend server pool changes, which is relatively infrequent. We then require that any time a server receives a packet that neither establishes a connection nor belongs to a connection it is already handling, it forwards the packet to the correct backend server for the connection. This requires servers to know the connection table mapping, which can either be learned from a centralized controller or by exchanging messages with other backend servers. This mechanism imposes additional latency in the worst case – *i.e.,* when the set of backend server changes – and requires additional communication between the servers, but assuming a stable deployment (*i.e.,* one where the backend pool changes infrequently) it performs as well as SilkRoad. Finally, similar to the previous scheme (where connection state was carried in packets headers), it imposes no limits on the number of connections that can be handled by a switch (because in both cases the switch has no connection state).

2.3 Enabling General Policies

The scalable schemes presented so far – one where the connection state was carried in packet headers, and one where the connection state was stored in servers – have looked at efficiently implementing load balancing based on consistent hashing, and do not address the question of more complex policies. We now show that the first mechanism can also be used to implement more complex load balancing policies including ones that depend on backend server load or on information about other connections. The main insight here is that while switches cannot implement general policies, such policies can be implemented in servers. Furthermore, since the connection mapping is carried in packets, only the initial packet needs to be processed by the policy server. Therefore, to implement more complex policies, one merely needs to configure the switch to forward the initial packet for each connection to a designated policy server, which is then responsible for forwarding the packet to an appropriate backend server. This can be done without switch state, as initial packets can be identified, *e.g.,* by the TCP SYN flag. Subsequent packets in the connection carry connection state, which the switch can use to directly forward packets to the appropriate backend server. While this scheme adds additional latency when processing new connections, it does not have any impact on throughput or latency beyond the first packet for a connection. Furthermore, while policy complexity in this case affects the rate at which new connections can be accepted, it does not have an impact on data throughput nor on forwarding scalability.

In summary, the three solutions presented above show that designs which restrict their use of switches so they are mostly stateless forwarding engines, with state and complex computation pushed to the end hosts, can provide the same performance as schemes which move all processing and state to programmable switches. Furthermore, separating the implementation in this manner ensures that performance does not come at the cost of limitations on application semantics, *e.g.,* all three designs presented here can scale to handle an arbitrarily large number of connections, and the third design (the one employing a policy server) can handle complex policies including ones that require analyzing state across connections.

Finally, while we did not fully elaborate on these issues, limiting switch state also has benefits for both fault tolerance – since state does not need to be synchronized across switches – and scaling – since adding additional load balancing capacity merely requires adding more switches.

3 LOAD PARTITIONING

3.1 Background

Load balancing provides a mechanism for scaling services where consistency is not a concern – either because connections do not change state, or because the application is designed to operate correctly despite inconsistency (*e.g.,* because it lacks shared state). However, this is not true for all services – services such as key-value stores which provide consistency often require that all requests for a given key go to one server or a small subset of servers – *i.e.,* they require load to be distributed according to actual requests. We refer to the problem of request-aware load spreading as *load partitioning*, and consider the implementation of such a service in the context of consistent sharded key-value stores.

Key-value stores often shard storage by using a consistent hashing scheme to decide where to place a key, which allows load to be distributed across servers without requiring a directory or some other datastructure to look up where keys are stored. However, as we discussed for load balancing, this can result in the most loaded server serving $\Theta(\frac{\ln n}{\ln \ln n})$ keys. Furthermore, most workloads exhibit a skewed access pattern, with some keys being accessed much more frequently than others. As a result, an individual server might be overloaded despite the use of random load distribution using consistent hashing.

NetCache [23] is a recent proposal for addressing this problem. NetCache relies on a previous observation [14] that shows that a relatively small cache, whose size is a function of the number of servers, is sufficient for eliminating load imbalances in a sharded key-value store. NetCache uses programmable switches to implement caching as follows: the switch is programmed so it can parse packets to determine the type of request (*i.e.,* whether a key needs to be read or written) and the key being accessed. On receiving a read request, the switch increments a counter that it uses to track the rate at which a key is accessed, and then checks to see if it has cached the item. If so, the switch generates a response with the appropriate value; if not, it forwards the request to the appropriate server. On receiving a write request, the switch checks to see if the key is cached, evicts it if so, and forwards the request to the appropriate backend server. A controller periodically collects access statistics from the switch and manages the switch cache so that frequently accessed items can be served from the cache.

While caching hot keys in a top-of-rack switch does improve load imbalance across key-value store shards, it also places significant limits on the semantics offered by the store including limits on the key size (which must be 16 bytes or less due to restrictions on packet header length) and value size (which cannot exceed 128 bytes due to limits on the amount of state that can be accessed while processing a packet). Furthermore, limits on switch memory mean that NetCache must rely on approximate datastructures – such as sketches – to store key access statistics, placing limits on cache policies that can be implemented by the controller.

3.2 An Alternative Approach

Similar to our observation for load balancing (§2), we find that by intelligently partitioning processing between switches and servers, we can retain the performance benefits of NetCache while simultaneously allowing arbitrarily large keys and values and arbitrary caching policies. To do so, we make two changes:

- First, we collect key access statistics on servers rather than collecting these on the switch. This allows us to collect accurate access statistics rather than relying on approximations.
- Second, rather than caching at the switch, the controller replicates overloaded keys on another server (or set of servers) – which might be chosen either randomly or by selecting the least loaded server(s) – and adds a rule to the switch indicating all servers at which the key is cached.

With these changes in place, the switch and servers process requests as follows. When the switch receives a read request, it checks to see if it has a rule for where the key is replicated, in which case it picks one of the replicas at random. If not, the switch hashes the key and forwards the request to the appropriate server. When the switch receives a write request it evicts any rules that refer to the key before forwarding the request to the correct server. Finally, we note that these changes by themselves do not address the limitation on key lengths. To address this, we adopt a solution inspired by NetCache – we require requests to include not just raw values for the key but also hashes. We limit the hash to be no longer than 16 bytes, thus allowing the switch to route requests without imposing limits on key length. Arguments similar to the ones used by NetCache ensure that at any given time the switch needs to hold rules for only a constant number of replicated keys, and ensure that this constant number of rules is sufficient to alleviate load imbalance across servers.

When compared to the original NetCache design, our approach adds a small amount of additional latency when accessing keys that are cached. Furthermore, assuming that total aggregate demand on the key-value store does not exceed the capacity of all servers, our approach can provide the same throughput benefits as NetCache.

It is useful to separate two possible rationales for using NetCache, and respond to them separately. First, there is the concern that the load can be imbalanced due to either the $\Theta(\frac{\ln n}{\ln \ln n})$ of random hashing, or to imbalances in key request patterns in read-dominated workloads. For this, the server-based approach described here, which uses the switch not to serve requests but to send requests to servers designed to handle the overload, has comparable performance.

Second, one can consider the case of extreme load imbalances that are write-dominated, where the rate of writes for one hot key exceeds the capacity of the server handling that key. Here, neither of the proposed solutions (NetCache and ours) can handle the load, as they both rely on a single server to handle writes.

4 LESSONS LEARNED

In the two cases we examined, load balancing and load partitioning, for the cases where application logic was implemented in programmable switches, we were able to find designs that were roughly comparable in performance and did not need application logic to reside in switches. Instead, they relied on a partitioning of functionality where all application logic (complex processing and state management) was handled by servers, and switches handled forwarding. However, the programmability of the switches was crucial, because this forwarding could require using nonstandard fields in the packet header. Thus, we would argue that moving application logic to switches is not *necessary* in these cases. But is it desirable? This is the question we address in this section.

Performance has been the main impetus for moving application functionality to programmable switches, since switch ASICs are designed to process terabits of aggregate traffic every second – which requires that they be able to process billions of packets per-second – providing an order-of-magnitude or more improvement in throughput when compared to software implementations. However, this gap between switch and software performance is a result of limitations on ASIC functionality including:

- Limits on how much of the packet can be accessed by the ASIC.
- Limits on the amount of state that can be stored in the switch.
- Limits on the amount of state that can be accessed while processing a single packet.
- Limits on the amount of processing that can be performed on an individual packet.
- Limits on the type of processing – *e.g.,* switch ASICs place restrictions on sharing data across packets matched by different rules in order to enable parallelism.

As a result, implementing application logic in ASICs requires imposing limits on application semantics. For example, NetCache must limit key and value sizes due to limitations on how much of the packet can be parsed and the amount of processing that can be performed on the packet. In addition, SilkRoad can only support a limited number of connections per switch due to limitations on state stored within a switch. Thus, while these switches have mind-boggling performance in isolation, there are important reasons that using them for sheer performance is misguided.

First, the limitations cited above mean that they can only be applied in very limited settings. If one needs larger value sizes, or needs to support many more connections, then these switch-based solutions do not degrade gracefully but instead "fall off a cliff". Thus, moving application logic to programmable switches is likely to impose stringent requirements that may interfere with the evolution of the application.

One might argue that we have no other option but to avail ourselves of switch hardware. However, in the two cases we examined, we did indeed find options with comparable performance. The key to doing so was to use the switch programmability to flexibly parse and steer packets, but to not involve them in the actual application logic.

Thus, our "lesson learned" is that it is neither appealing nor necessary to constrain application functionality so that it can be implemented in switch ASICs. Instead, we advocate refactoring these applications to use hardware as intended – *i.e.,* use switches for forwarding and routing messages in a way that requires minimal state, and use general purpose servers to handle stateful general purpose computation. Also, if the primary consideration in moving application logic to switch ASICs is the ability to process terabits of aggregate application traffic, then acceleration devices like FPGAs and GPUs might provide better solutions; these devices can be specialized to provide the appropriate amount of state and computing that is desired by an application as opposed to repurposing the switching hardware to suit application logic. In fact, such devices can also be co-located along with the switch if reducing the processing latency is a consideration. All of these implementation and deployment options are consistent with our observation that applications should use hardware as intended.

While our observations were drawn from an analysis of load balancing and load partitioning, they also hold for other classes of applications, which we comment on briefly below:

- **In-network aggregation** DAIET [32] is a recent proposal for performing aggregation (as performed by reducers in map-reduce jobs or parameter servers in ML training) using programmable switches. This is beneficial for performance since all transfers within a rack must traverse the switch – thus minimizing latency – and switches are provisioned so they can simultaneously receive traffic at line rate from all ports – thus avoiding incast and other congestion concerns. However, the use of switches for aggregation severely limits the types of algorithms that can be used when aggregating information. Existing techniques in the HPC community, *e.g.,* butterfly allreduce [36] and variants, already address the incast problem for aggregation. While it is impossible to match the latency benefits of DAIET, our research on the topic and conversations with practitioners suggests that co-locating compute accelerators along with switches would allow for both generality and performance in the long term.
- **In-network coordination** Proposals including NetPaxos [8], NetChain [22], and others have suggested implementing consensus in programmable switches. While these implementations do improve the rate at which locks can be acquired or messages can be sequenced – generally by reducing latency for sending and processing consensus messages – they do not speed up the actual processing that needs to be performed in the critical section (in cases where a lock is acquired) or the rate at which messages are processed (in cases where messages are sequenced). As a result these proposals have only limited impact on application performance. In this case, we believe that while it is reasonable to implement simple stateless primitives such as consensus in switches, this is neither necessary nor useful in isolation. Instead, systems incorporating these proposals need to first make algorithmic advances or leverage other hardware acceleration to improve other parts of the application before focusing on improvements to consensus algorithms.

5 GOING BACK TO NETWORKING

While moving application functionality into switches is both unnecessary and unhelpful, programmable switches do enable the development of new types of network functionality, such as the examples listed below:

- **Network Load Balancers** Systems such as Conga [1], have suggested that congestion-aware load balancing might be used to select routes for TCP flowlets in datacenters. The load balancing algorithm used by Conga and similar systems requires keeping track of the number of packets sent out on any port and incorporating this into congestion information from the path. These statistics change frequently (*i.e.,* every time a packet is sent out) and the accuracy of this algorithm depends on the granularity at which congestion information can be updated. As a result, these updates are best performed in switches, which are on-path and already have visibility into aggregate traffic metrics for each port, and the load balancing policy itself must be implemented on the switch so it can have low-latency access to this information. Finally, while state exhaustion is also a concern in this case [33] – since the switch must maintain a connection table like datastructure to track individual flowlets – this can be addressed by embedding such state in packets as suggested in §2.
- **Network Telemetry** Recent systems such as Marple [26] have also proposed executing complex network queries on programmable switches. Here again the main insight is that queries in general require fine grained access to switch statistics, which cannot be easily shipped to a server. Instead, executing queries on switches allows these statistics to be aggregated before being shipped to a server, thus limiting the amount of control bandwidth required. Note that the cardinality of queries supported by such systems is limited due to limitations on switch memory and number of registers, but similar limitations apply to any other approach.
- **Packet Scheduling** Packet scheduling, which determines the order in which packets are processed by a switch, is an oft-cited mechanism for enabling more efficient sharing of link capacity between users, and better congestion control. However, switches traditionally only implement a few fixed packet scheduling algorithms. Recent proposals such as PIFO [35] have looked at frameworks that can be used to implement a variety of packet scheduling algorithms on programmable switches, and other packet scheduling proposals such as approximate fair queueing [34] are specifically designed to account for the capabilities offered by programmable switches. Functionally, implementing such frameworks or algorithms outside of a switch is meaningless and hence they must be implemented within switches regardless of implementation limitations. Thus, developing programmable switches where these could be deployed would be of great value.
- **Congestion Control** Prior work such as RCP [11] has also shown the benefits of implementing switch-based congestion control mechanisms that set rates based on measured link utilization. Similar to network telemetry systems, these mechanisms require access to fine grained switch counters, and hence also must be implemented in switches.

As can be seen, these and other similar functions need timely access to fine-grained packet counters so they can rapidly adjust routing decisions in response to network state, and hence must be placed within the network. Programmable switches greatly simplify the deployment of such solutions, and while these solutions do suffer from the same limitations mentioned in §4, in this case no host based solution is available to us.

In addition, the functionality provided by these proposals is not limited to a particular application but instead affects all (or most) network traffic, and hence they are designed to be deployed by the network operator or architect, who can engineer the deployment to avoid resource exhaustion problems. For example, Conga deployed on a top-of-rack switch benefits from the limited number of active TCP connections in the rack, thus avoiding concerns about running out of space for tracking flowlets. Designing networks to ensure protocol scalability and avoid resource exhaustion is not unique to programmable networks, but the main observation here is that in this case network operators only need to consider the needs of the network and not of all applications.

Thus, we would amend our "Lesson Learned" to add that while putting application logic into programmable switches is problematic (for all the reasons we previously discussed), implementing networking-related solutions where no host-based alternative is available has no such downsides.

6 CONCLUSION

While programmable switches provide a powerful primitive for specializing networks for specific applications, they are not a panacea for improving application performance. Some recent papers have focused on finding ways to shoehorn application semantics into the limited capabilities of programmable switch ASICs. This has demonstrated great cleverness, as overcoming the resource restrictions inherent to ASICs is no easy task. However, we question whether this approach is wise.

In particular, in this paper we showed that moving application logic into programmable switches is both unnecessary – since similar performance can be achieved by offloading a much smaller portion of application semantics – and harmful – since such shoehorning often imposes constraints on application behavior (*e.g.,* limiting the size of values that can be handled) or, worse yet, limits application semantics. As a result, we observe that when it comes to implementing application specific functionality, switches should continue to be used as switches, *i.e.,* as network devices that receive packets, parse them, and then forward them with little or no modification.

7 ACKNOWLEDGMENT

We thank Aisha Mushtaq for being an invaluable participant in several early discussions on load balancing, and Olivier Bonaventure for his insightful comments and suggestions on this editorial. This work was funded in part by an early career faculty grant from VMware, a grant by Intel Corporation and by NSF grants 1616774, 1714508, 1704941, 1817115, and 1817116.

REFERENCES

[1] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. Conga: distributed congestion-aware load balancing for datacenters. In *SIGCOMM*, 2014.
[2] Arista. Arista 7170 Series. https://www.arista.com/en/products/7170-series (Retrieved 07/19/2018).
[3] Barefoot Networks. Product Brief: Berfoot Tofino. https://barefootnetworks.com/products/brief-tofino/ (Retrieved 07/19/2018).
[4] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger. Tcp extensions for high performance, September 2014. RFC7323.
[5] Cavium. XPliant Switch Product Family. https://www.cavium.com/xpliant-ethernet-switch-product-family.html (Retrieved 07/19/2018).
[6] Cisco. Cisco Catalyst 9000 Family Switches. https://www.cisco.com/c/en/us/products/switches/catalyst-9000.html (Retrieved 07/19/2018).
[7] S. P. Cole. New Trident 3 switch delivers smarter programmability for enterprise and service provider datacenters. https://www.broadcom.com/blog/new-trident-3-switch-delivers-smarter-programmability-for-enterp (Retrieved 07/19/2018), 2017.
[8] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. Netpaxos: consensus at network speed. In *SOSR*, 2015.
[9] Y. Desmouceaux, P. Pfister, J. Tollet, M. Townsley, and T. H. Clausen. 6lb: Scalable and application-aware load balancing with segment routing. *IEEE/ACM Transactions on Networking*, 26, 2018.
[10] F. Duchene and O. Bonaventure. Making multipath tcp friendlier to load balancers and anycast. *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–10, 2017.
[11] N. Dukkipati and N. McKeown. Why flow-completion time is the right metric for congestion control. *Computer Communication Review*, 36(1):59–62, 2006.
[12] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI*, 2016.
[13] F5 Networks. BIG-IP Platform. https://f5.com/products/big-ip (Retrieved 07/19/2018).
[14] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: provable load balancing for randomly partitioned cluster services. In *SoCC*, 2011.
[15] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. FranÃğois. The Segment Routing Architecture. *2015 IEEE Global Communications Conference (GLOBECOM)*, 2014.
[16] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. Tcp extensions for multipath operation with multiple addresses. *RFC*, 6824:1–64, 2013.
[17] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, and C. Paasch. TCP Extensions for Multipath Operation with Multiple Addresses – draft-ietf-mptcp-rfc6824bis-11. RFC 6824bis Draft, IETF, May 2018.
[18] Intel. Programmable ASICs: Flexibility at Wire Speed on TechWiseTV. https://www.cisco.com/c/m/en$_u$s/training-events/events-webinars/webinars/techwise-tv/214-programmable-asics.html (Retrieved 07/19/2018).
[19] Intel. Intel Ethernet Switch FM6000 Series. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf (Retrieved 07/19/2018), 2012.
[20] T. Jepsen, M. Moshref, A. Carzaniga, N. Foster, and R. Soulé. Life in the fast lane: A line-rate linear road. In *SOSR*, 2018.
[21] T. Jepsen, M. Moshref, A. Carzaniga, N. Foster, and R. Soulé. Packet subscriptions for programmable asics. In *HotNets*, 2018.
[22] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. Netchain: Scale-free sub-rtt coordination. In *NSDI*, 2018.
[23] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *SOSP*, 2017.
[24] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. R. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. The quic transport protocol: Design and internet-scale deployment. In *SIGCOMM*, 2017.
[25] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *SIGCOMM*, 2017.
[26] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *SIGCOMM*, 2017.
[27] V. Olteanu and C. Raiciu. Datacenter scale load balancing for multipath transport. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, pages 20–25, New York, NY, USA, 2016. ACM.
[28] V. A. Olteanu, A. Agache, A. Voinescu, and C. Raiciu. Stateless datacenter load-balancing with beamer. In *NSDI*, 2018.
[29] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. Netbricks: Taking the v out of nfv. In *OSDI*, 2016.
[30] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, et al. Ananta: Cloud scale load balancing. *Computer Communication Review*, 43(4):207–218, 2013.
[31] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2:277–288, 1981.
[32] A. Sapio, I. Abdelaziz, M. Canini, and P. Kalnis. Daiet: a system for data aggregation inside the network. In *SoCC*, 2017.
[33] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. S. Nelson, and S. Peter. Evaluating the power of flexible packet processing for network resource allocation. In *NSDI*, 2017.
[34] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *NSDI*, 2018.
[35] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *SIGCOMM*, 2016.
[36] H. Zhao and J. F. Canny. Sparse allreduce: Efficient scalable communication for power-law data. *CoRR*, abs/1312.3020, 2013.