Adaptive Weighted Traffic Splitting in Programmable Data Planes

Kuo-Feng Hsu
Rice University

Praveen Tammana
Princeton University

Ryan Beckett
Microsoft Research

Ang Chen
Rice University

Jennifer Rexford
Princeton University

David Walker
Princeton University

ABSTRACT

Recent work introduced load-balancing algorithms that dynamically pick the best path *entirely in the data plane*, to react to traffic dynamics on a small timescale. This paper takes the next step to balance load dynamically across *multiple* paths in the data plane. The design of such a load-balancing primitive raises interesting challenges due to the hardware constraints of the data plane. We show that these constraints create practical problems for Weighted-Cost MultiPath (WCMP), which replicates hash-table entries in proportion to the weight of each path. Under these hardware constraints, naïve implementations of WCMP take a long time to converge to new weights. We then present a hash-based data structure that achieves adaptive traffic splitting in programmable data planes. Our data structure carefully partitions the arithmetic operations required to a) split traffic in proportion to the path weights and b) update the path weights, by leveraging a multi-stage pipeline and stateful ALUs. By doing so, accurate splitting and efficient updates are done at line rate. We implement our data structure in P4 and our preliminary evaluation shows significant reduction in flow completion time compared to other data-plane load-balancing schemes such as HULA.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **Network management**; **Traffic engineering algorithms**.

1 INTRODUCTION

Network operators typically provision multiple paths to meet traffic demands of their applications, whether in datacenters or wide-area networks. However, utilizing available bandwidth requires an effective way to balance the traffic load, especially when the load on these paths can change quickly due to traffic fluctuations. Therefore, load balancing on a small timescale based on traffic conditions of the paths becomes crucial for achieving good performance.

Traditional switches have limited support for load-aware traffic splitting. Mechanisms like Equal-Cost MultiPath (ECMP) and Weighted-Cost MultiPath (WCMP) with static path weights cannot adapt their splitting strategies to the traffic conditions. The recent emergence of programmable switches [1, 8] has opened up a new range of possible solutions in the data plane while maintaining high packet-processing throughput. The switches have multiple hardware stages for parallel processing, which can achieve Tbps throughputs. In addition, each stage can maintain state across packets in persistent memory, and the state can be accessed or updated based on results carried by a packet from previous stages.

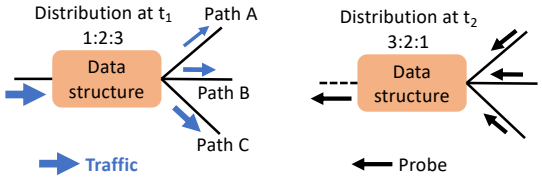


Figure 1: DASH aims to performs adaptive weighted traffic splitting in programmable data planes.

Recent work [5, 15, 20] has seized the opportunity to develop load-balancing solutions that can adapt to traffic conditions entirely in the data plane. For instance, CONGA [5] leverages customized switch hardware to achieve this; HULA [20] and Contra [15] are designed for P4-based programmable switches. These solutions update the data plane state and direct traffic in fine timescales based on traffic conditions. However, one common limitation of these solutions is that they only consider use *a single "best" path* at any given time. The benefits of using multiple paths have been demonstrated by many projects (*e.g.*, TRUMP [12], HALO [22], TeXCP [17], MATE [10]), most of which process probes that carry path status information in the control plane and generate the updates of the data plane state. It remains unclear how (or whether) equivalents of these control plane solutions can be realized in practice using commodity data planes.

In this paper, we set out to bridge the gap between these two classes of solutions by proposing new data structures for load-aware traffic splitting in the data plane. As illustrated in Figure 1, these new data structures simultaneously (i) spread new flows across multiple paths in proportion to the current load on those paths, and (ii) dynamically adjust to load changes at data-plane speeds. Doing so requires the design of new data structures that can (i) assign a new flow to a path according to the current distribution D, where D is a set of path weights, and (ii) update the distribution D after processing path status information carried in data-plane probes (more details in section §2).

The goal of supporting load-aware traffic splitting gives rise to a set of interesting design techniques, which are needed to address the hardware constraints of today's programmable data planes. At a high level, programmable data planes have the following constraints:

- Limited number of per-packet accesses to register memory within a single stage.

SOSR '20, March 3, 2020, San Jose, CA, USA

ACM ISBN 978-1-4503-7101-8/20/03…$15.00
https://doi.org/10.1145/3373360.3380841

- Register memory in one stage cannot be accessed by a packet at other stages, to avoid hazards caused by concurrent accesses by packets at different stages.
- There are a fixed number of stages to guarantee low per-packet latency. The total number of per-packet operations is constrained by what can be performed in these stages.

Contributions. In this work, we (1) characterize the tradeoffs for implementing data structures under the constraints, (2) study the tradeoffs of different novel data structures we have created, and (3) then focus on one data structure that works well in a wide range of practical scenarios.

One popular splitting data structure is WCMP that replicates hash-table entries with path IDs in proportion to the weight of each path in a distribution D. A key advantage of WCMP is there is no limit on the number of paths supported, as long as the table size is big enough to keep the error between actual entries and desired entries reasonably low. But this advantage comes at the cost of either slow update or extra overhead. However, updating the distribution D using WCMP requires modifying the path IDs for many table entries, which is infeasible due to the limited number of per-packet memory accesses in each stage. Thus, the update process takes a long time, which in turn causes deviation from the desired traffic splitting (more details in section §3.2). As an alternative, we could divide the required modifications to the table across packets. But keeping track of the modifications requires sequentially reading multiple registers across multiple stages, and then writing to the registers. Since the read-write to a register from different stages cannot be supported, it would require packet recirculation, leading to inefficiency (more details in section §3.3).

Our proposal works very differently from WCMP-based algorithms. By assigning a unique region in the hash function output space to each path, where the region size is in proportion to the weight of each path in a distribution D, we can do fast and efficient update to D. In contrast to modifying many table entries, now the update requires modification to a small number of region boundaries. By carefully assigning the boundaries to switch pipeline stages, we can significantly reduce the number of memory accesses required. Also, this would avoid the need for accessing the registers of a stage from different stages. As a tradeoff, this comes at the cost of the number of paths supported, as the number of stages and number of per-packet operations in each stage are limited (more details in section §4).

The basic idea of partitioning the hash space into unique regions is not new itself—and indeed may appear quite natural—but the realization that we can tilt the data structure sideways in the data plane (and handle a small set of regions per pipeline stage) is new. Based on this idea, we present **D**ata-plane **A**daptive **S**plitting with **H**ash threshold (**DASH**) that leverages a multi-stage pipeline and per-stage stateful ALUs and update the distribution D at line rate, thus enabling quick adaptation to traffic conditions and efficient utilization of network capacity. We evaluate the performance of DASH in terms of flow completion time and compare with other data-plane load-balancing schemes such as HULA and ECMP, and found that DASH can improve the flow completion time significantly.

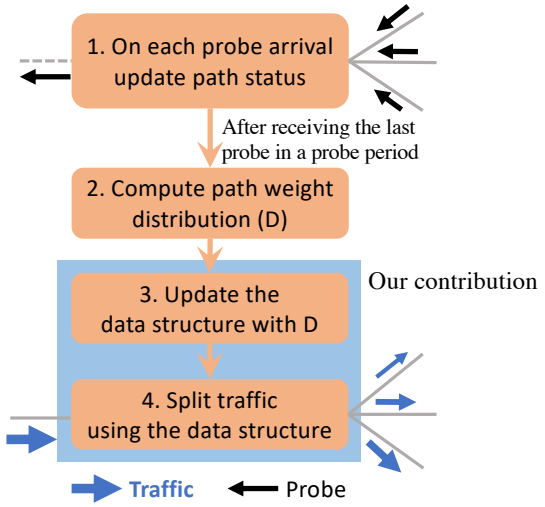


Figure 2: Adaptive traffic splitting

Algorithm	Efficiency	Accuracy	Cardinality
Iterative (§3.1)	✗	✓	✓
Probabilistic (§3.2)	✓	✗	✓
Deterministic (§3.3)	✗	✓	✓
DASH (§4)	✓	✓	✓–

Table 1: Design goals met by weight update algorithms

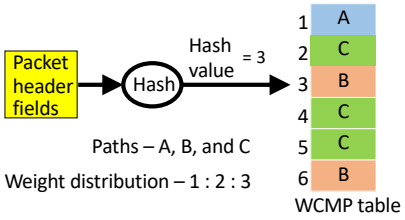
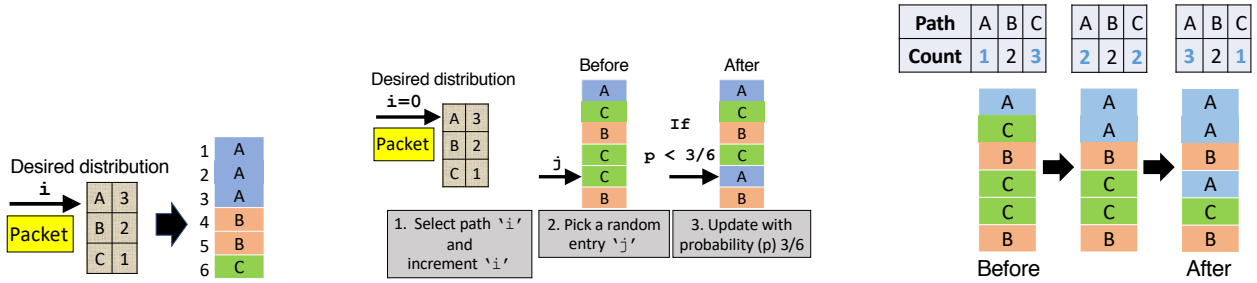


Figure 3: Packet-to-Path mapping using WCMP table

2 ADAPTIVE TRAFFIC SPLITTING

Computing weight distribution (D). We assume that probes are generated by destinations (*e.g.*, hosts, switches) periodically on the order of RTTs using out-of-band mechanisms (*e.g.*, HULA [20], Contra [15]). These probes carry path performance metrics (*e.g.*, utilization, delay) derived from high-level policy goals (*e.g.*, minimize latency, maximize throughput). As shown in Figure 2, after receiving the last probe in a probe period, one could compute the distribution D for each destination using a simple heuristic that sets the weight to be inversely proportional to the path utilization—*i.e.*, the utilization of the bottleneck link on the path that a probe traverse. Similar to HULA [20], the switches update the utilization stored in the probe at every hop with the maximum of a) utilization of the original probe, and b) utilization of the inbound link on which the probe has arrived. We could also use other weight computation schemes designed to run at every hop [11, 22, 23] or at the edges [10, 17] which aim to reduce traffic sent on expensive paths and increase the fraction on the best path. In addition, they keep the network stable by not overreacting to the load changes.



(a) **Iterating over WCMP table** (b) **Probabilistically update the WCMP table** (c) **Deterministically update the WCMP table**

Figure 4: WCMP-based weight update algorithms

Data structure input. In this paper, we assume that the path weight distribution D for each destination is provided by one of the above schemes, leaving the question of their implementation as future work. For instance, consider the scenario in Figure 1, where there are three links, each belonging to a different path towards a destination. At time t_1, D is 1:2:3 (current). At time t_2 probes on all three paths arrive and the distribution is updated to 3:2:1 (desired). Now, the main challenge is updating the underlying data structure at line rate under the hardware constraints such that the distribution of traffic most closely approximates the desired distribution. More specifically, consider a case where the time taken to update the data structure after processing probes is in the order of milliseconds (msec). On a switch processing traffic at 1 Tbps, every 10 msec delay would cause approximately up to 1 GB of traffic to not follow the desired distribution D. Therefore, for load-aware traffic splitting, fast response to new path weights is critical.

Design goals. Given the desired D, we focus on designing a traffic splitting data structure that can do (i) accurate splitting (accuracy), (ii) efficient update (efficiency), and (iii) support many paths (cardinality); having data structures that do one or two but not the others is not hard, but achieving all three simultaneously is hard. In the following sections, we present different data structures and their tradeoffs for implementing the weight update algorithms; Table 1 summarizes the design goals met by each algorithm.

3 WCMP-BASED ALGORITHMS

In the section, we discuss tradeoffs of three weight update algorithms based on WCMP. To represent the path weight distribution D, WCMP replicates hash-table entries with the same path ID in proportion to the weight of each path. When a packet arrives, the switch computes a hash value from the packet header fields and sends the packet along the path ID at the table entry indexed by the hash value. For example, consider the scenario in Figure 3, where there are three paths (A, B, and C) with weight distribution 1:2:3, and the distribution is represented in the table with 1, 2, and 3 entries respectively. The probability that a packet is sent along path B is 1/3.

Problem statement. Assuming that:

- There are K paths, i=0, 1,..K-1;
- A WCMP table of size N;
- Current number of entries assigned to the i-th path is $\{C_i\}$; and
- Desired number of entries for the i-th path is $\{T_i\}$;

where $\sum_{i=0}^{K} C_i = \sum_{i=0}^{K} T_i = N$

The goal of an algorithm is to update the WCMP table such that the error (e) between the desired entries and the current entries is minimum. Error (e) is defined as $\frac{1}{2N} \sum_{i=0}^{K} |T_i - C_i|$.

3.1 Iterating over the WCMP table

One strawman algorithm for updating the table would be to iterate over K paths upon a packet's arrival, and for every path i allocate entries in the WCMP table until the number of entries of i is equal to T_i. Figure 4(a) illustrates this solution. However, this algorithm requires loop constructs and access to many memory locations to update the table entries, which make the algorithm challenging to perform line-rate packet processing without recirculation. Therefore, this strawman solution is inefficient (see §2).

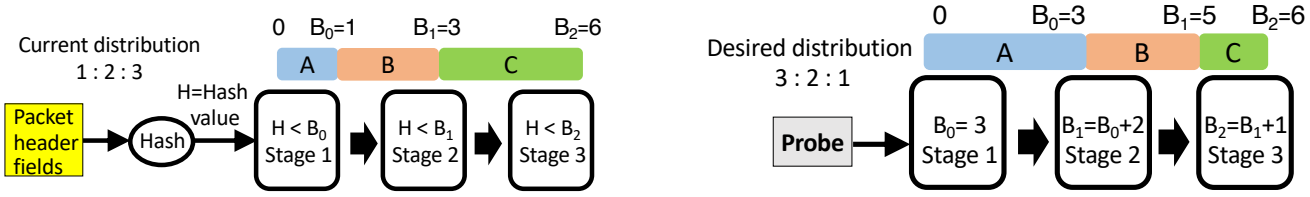
3.2 Probabilistic WCMP table updates

To overcome the limitations of the iterative approach, an alternative is to divide required modifications across packets, such that each packet would do a small number of updates to the WCMP table. One possible way to realize the idea is to have every packet (i) pick a path i in round-robin fashion, (ii) pick a table entry j randomly, and (iii) assign the entry j to path i with probability proportional to the path's desired weight (see Figure 4(b)). Specifically, if a random value r ranging from 0 to $N-1$ is less than desired number of entries T_i, then the entry j is assigned to the path i.

Though the algorithm is simple and implementable, there are two problems that increase the time for it to converge to the desired number of entries (T_i). First, if path i needs a small fraction of the entries, then the packet is most likely not used to update the WCMP table. For example, consider that path i needs 1 out of 10 entries, then a packet misses an opportunity to update the table for 90% of the time. Second, the random selection of the WCMP entry j may steal an entry from a path that is already in deficit (*i.e.*, $C_j < T_j$), causing further deviation from the desired distribution.

The expectation of the current entries C_i is Exponentially Weighted Moving Average (EWMA) over the desired entries T_i with a coefficient $\alpha = \frac{1}{N}$; when N is large, α is small, thus convergence is slow. We omitted the proof due to space limitations. Suppose K=32, N=3200, and error=5%, we need up to 250K packets to converge. For a switch processing 1 KB packets at 1 Tbps rate, this means up to 250 MB of traffic does not follow the desired distribution, and this grows as K and N increase.

In summary, the non-deterministic amount of time to reach the desired distribution makes the probabilistic approach inaccurate (goals in §2).



(a) **Packet-to-Path mapping using DASH**

(b) **DASH weight update procedure**

Figure 5: DASH data structure

3.3 Deterministic WCMP table updates

As an alternative, a deterministic approach would assign entries for a path that has more entries (i.e., non-deficit path) to another path that does not have enough (i.e., deficit path). This can be done by maintaining a per-path counter that keeps track of the cumulative effect of past assignments of entries to paths (see Figure 4(c)). This approach first identifies deficit and non-deficit paths based on their counter values and then move entries from non-deficit paths to deficit paths.

For re-allocating entries of a non-deficit path to a deficit path, we need to maintain three variables in registers: the current path i, WCMP entry j, and the current number of entries C_i. Updates to i, j, and C_i are done as follows. (1) If the path i is non-deficit ($C_i \not< T_i$), move current i immediately to another path. This will significantly reduce the number of packets not used to update the table (see §3.2), therefore reducing convergence time. (2) If the path at entry j is in deficit ($C_j \leq T_j$), then move j to the next entry to avoid reducing the number of entries from the path that is already in deficit. (3) Finally, if none of the above holds, then we reallocate the entry j to the path i—i.e., incrementing C_i and decrementing C_j both by 1.

Though fast, this algorithm is complex as it needs to perform read-write on multiple registers that spread across multiple stages, thus it cannot be implemented efficiently without packet recirculation [1]. For instance, consider the code that updates i in the step 1: (a) `read` i; (b) `read` C_i, T_i; and (c) `If` $C_i \not< T_i$ `then` $i = i$+1. Since the reads are dependent, they cannot be executed in the same stage. But, if the condition $C_i \not< T_i$ which is executed in the later stage becomes true, the packet cannot write to i's memory, because the memory is bound to the previous stage. However, we could finish the read-write by recirculating the packet with the updated i value. Nevertheless, recirculation is expensive because a switch can handle a fixed number of packets per second and recirculating a packet multiple times reduces packet-processing throughput. To understand the recirculation overhead, the upper bound on the required number of packets for the algorithm is $N + K$. So, for K=32, N=3200, error=0%, it is about 3000; significantly reduced compared to the probabilistic algorithm. But, suppose the probe period is 60 msec, then the recirculation overhead in each probe period is roughly 50K packets per second (PPS), and this increases as the probe period decreases, especially in low-latency networks with RTTs in the order of microseconds.

Therefore, the large amount of traffic recirculation makes this deterministic approach very inefficient (goals in §2).

4 THE DASH DATA STRUCTURE

To overcome the efficiency and accuracy limitations of the previous two approaches, we present DASH. Its key idea is that, instead of replicating entries in the WCMP table for each path, a unique region is assigned in the hash function output key space in proportion to the weight of each path (see Figure 5(a)). When a packet arrives, the hash value generated from the packet header fields is compared against the region boundaries and determine the region it matches and the associated path. Now, updating weights become simple, as we now need to update a small number of region boundaries instead of many table entries in the WCMP-based algorithms.

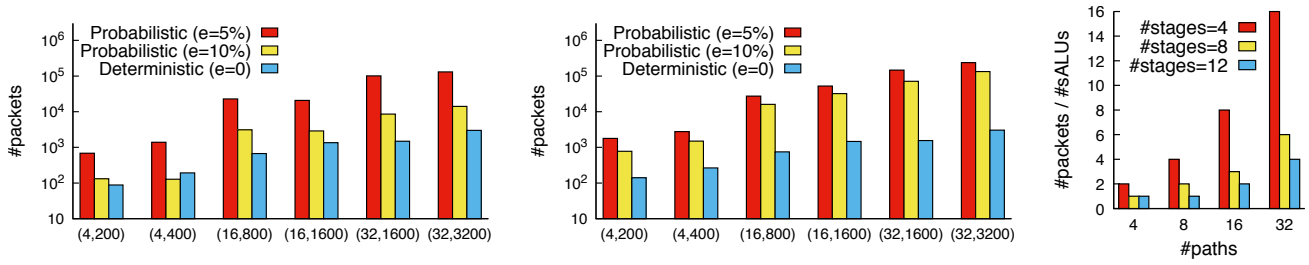
4.1 Assigning region boundaries

Packet-to-Path. Consider a scenario where there are three paths towards a destination. We assign one path and the corresponding boundary to one stage in a multi-stage pipeline. As shown in Figure 5(a), a path's boundary is stored in the register memory mappped to a stage, and the boundary is compared against a packet's hash value using a stateful ALU (SALU). For instance, if the packet's hash value is less than the path boundary, then the packet is sent along that path.

Updating boundaries. It becomes easy to update the path boundaries with just a single packet; we could use the last probe in a given probe period. The core idea is a path's new boundary is obtained by adding the path's region size to the previous path's boundary (see Figure 5(b)). For this, as the packet goes through the pipeline it carries the boundary of the last updated path in its metadata. Using a SALU we then add the path's region size to the metadata and the result of the addition operation (*i.e.*, the cumulative sum of region sizes) is updated at two locations: 1) in the stage register where path's boundary is maintained; and 2) in the packet's metadata carry forwarded to the next stage.

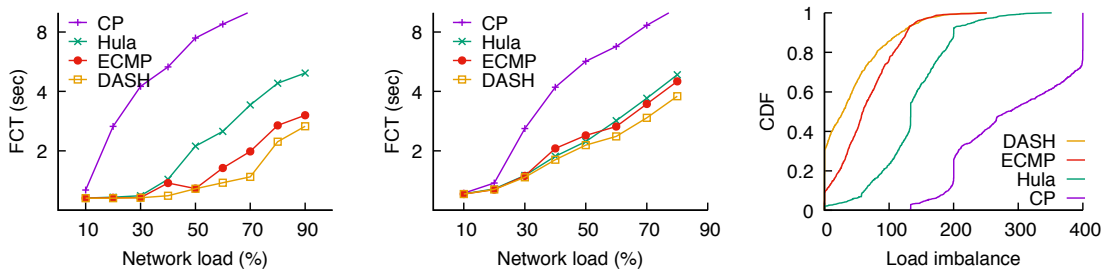
However, in this simple one-to-one assignment, the number of paths supported cannot go beyond the number of stages actually present or allowed to use for traffic splitting. For instance, in a pipeline with 12 stages, a maximum of 10 paths per destination is supported. The other two stages are used to compute the hash value and to retrieve the associated path[1]. Note that only one register per destination is used to maintain the boundary mapped to a stage. To support more destinations, we could simultaneously use the other stage registers.

[1]The packet metadata also has a bit for each region boundary that stores the comparison result (either 0 or 1). All bits in the result is used as a key to retrieve the associated path from a match-action table.



(a) **Gaussian distribution. (K,N) : (#paths, table size)** (b) **Bimodal gaussian distribution. e: error.** (c) **DASH resource usage**

Figure 6: DASH needs significantly fewer number of packets than the WCMP-based algorithms to update weights.



(a) **FCT for symmetric topology** (b) **FCT for Asymmetric topology** (c) **Load imbalance at 70% network load**

Figure 7: DASH achieves a significantly shorter FCT than HULA, outperforming ECMP considerably.

4.2 Leveraging concurrency within a stage

High cardinality. To support a greater number of paths per destination (*i.e.*, high cardinality), we can leverage the *multiple* SALUs in each stage to execute multiple parallel arithmetic operations (*e.g.*, comparisons, additions). By doing so, the cardinality is determined by the product of the number of per-stage SALUs and the number of stages in the pipeline. For example, with 10 stages[2] and 8 per-stage SALUs, the cardinality is 80 paths per destination. For 80 paths, DASH needs 8 per-stage SALUs and 8 32-bit per-stage registers to store the per-destination boundaries; the memory overhead is small compared to the today's ASICs, which have 11.5 Mb of SRAM [9] in each stage. This leaves plenty of memory resources for other functions (*e.g.*, access control, routing). The main bottleneck comes from the number of available SALUs, and should be resolved by better resource allocation techniques.

Updating boundaries. Updating all path boundaries (stored in per-stage registers) requires a long sequence of dependent operations. This is illustrated using Figure 5(b), where a path's boundary as mapped to a stage is obtained only after adding the path's region size to the boundary of the path mapped to the previous stage. To implement this, we can recirculate the probe packet with the last updated boundary in a given pass. For example, suppose stage 1 has 8 boundaries ($B_0, B_{10}, .., B_{70}$), stage 2 has $B_1, B_{11}, .., B_{71}$, and so on. With 10 stages in the pipeline, in each recirculation we can update only 10 boundaries: $B_0, B_1, .., B_9$ in the first pass, $B_{10}, B_{11}, .., B_{19}$ in the second pass, and so on. Therefore, the number of recirculations is equal to the number of SALUs in each stage minus one (*i.e.*, 7); this is significantly fewer than the WCMP-based algorithms.

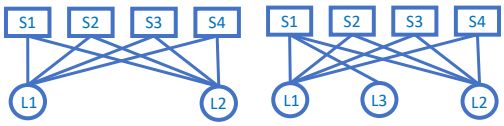
[2]RMT [9] contains 32 stages.

5 EVALUATION

Our main goal for evaluation is to understand how fast the DASH data structure can update its hash boundaries to realize new path weight distribution (D), and how this adaptive traffic splitting translates to performance improvements. We have built a prototype of DASH based on a customized version of ns3 [3] with P4 `bmv2` model support. As baseline, we have implemented the HULA [20] load-aware scheme that always picks a single best path for forwarding in the data plane. To evaluate the convergence time, we have implemented simulated versions of WCMP-based algorithms in Python. All experiments have been conducted on a Dell server with six Intel i7-8700 CPU cores, 16 GB RAM, running Ubuntu 16.04.

5.1 Convergence time

First, we measure the convergence time in terms of the number of packets it takes for the WCMP-based algorithms and for DASH to adjust to new path weights (*i.e.*, D). For the WCMP-based algorithms, we generate synthetic path weights following (1) Gaussian distribution with small variances ($N(4, 1)$) to represent uniform splitting over paths, and (2) Bimodal Gaussian distribution that picks the weights with equal probability between $N(4, 1)$ and $N(16, 1)$ to represent non-uniform splitting over paths. Figure 6(a) and Figure 6(b) show the convergence time for the different systems for different numbers of paths (K) and different WCMP table sizes (N); at a high level, as N and K become large, the required number of packets increases significantly. For small K and N (*e.g.*, K=4 and N=200), the deterministic algorithm requires up to 100 packets to keep $e = 0$, whereas DASH needs just 1 packet with 8 stages using at most 1 SALU in each stage (see Figure 6(c)); for K=32 and N=3200 it is 3000 packets and 8 packets, respectively. Therefore, DASH can adapt to new weights much faster than the WCMP-based algorithms.



(a) Symmetric topology (b) Asymmetric topology

Figure 8: Topologies

5.2 Performance improvements

Next, we evaluate how this faster adaptation leads to performance improvements. We set up two topologies: symmetric (see Figure 8(a)) and asymmetric (see Figure 8(b)). In both topologies, we have 32 hosts with 500 Mbps links. Half of these are connected to switch $L1$ as senders and the other half to switch $L2$ as receivers. $L1$ and $L2$ are connected with four 1 Gbps links to four other switches ($S1 - S4$). Hence it is a 2:1 oversubscription. To make it asymmetric, an extra switch $L3$ with 4 hosts below is connected to $S1$. Hosts generate traffic based on the web search workload in DCTCP [6], and the time gap between flow arrival follows a Poisson distribution (the mean is determined from the network load). We have tested several load-balancing mechanisms: 1) ECMP balances traffic randomly without considering network load; 2) HULA is load-aware and always sends traffic along the least-utilized path (the current best path); 3) the control plane solution (CP) adjusts path weights derived from the path utilization information in probes on large time scales (1 sec); and 4) DASH updates path weights with the inverse of path utilizations at the end of every probe period and splits new flowlets based on the new weights.

Flow completion time. Figure 7(a) shows that DASH is 12-25% better than ECMP in the symmetric topology because ECMP is not load-aware and continues sending traffic on congested paths. In comparison, DASH splits traffic in proportion to the load on each path, thus it shifts traffic from congested paths to other paths and achieves better performance. HULA does not perform well because it overloads the current best path while keeping the other paths underutilized. In the asymmetric topology, ECMP performance degrades and comes close to HULA, because ECMP continues sending traffic on the congested path ($L1$-$S1$-$L2$), whereas HULA can adjust to the path performance. At 80% workload, DASH outperforms Hula by 22.1% and ECMP by 16.0%.

Load imbalance. Next, we measured the load imbalance of each system by computing the throughput differences between the most and least loaded switch-to-switch links of $L1$ and normalizing them by the average throughput across the links. As shown in Figure 7(c), DASH balances traffic much better than the other schemes; ECMP load imbalance is greater than 80% for 35% of the time whereas for DASH it is only 20%.

6 RELATED WORK

Static traffic-splitting schemes, such as WCMP [24] MPLS-TE [2], Niagara [19], use table entries, thus updating the table in the data plane is inefficient. The idea of hash boundaries was initially proposed in [14], but hash boundaries are not used in practice as it requires hardware modifications. DASH showed how the programmable data planes make it possible to implement hash boundaries. DASH data-plane primitive complements P4-based traffic-splitting schemes like MP-HULA [7] which splits MPTCP subflows over K-best paths. On the other hand, centralized traffic engineering solutions (*e.g.*, B4 [16], SWAN [13], Hedera [4], SMORE [21]) are too slow to react to dynamic traffic conditions. In contrast, DASH complements distributed solutions (*e.g.*, CONGA [5], HULA [20], Contra [15], TeXCP [17], HALO [22], Gallagher [11]) by enabling quick adaptation to traffic conditions.

7 DISCUSSION

Online distributed traffic engineering. We envision a programmable system where operators specify high-level network-wide policies (e.g., traffic engineering, routing), and the system at runtime generates corresponding measurement and control operations to run in a fully distributed manner for load-aware traffic splitting. Such dynamic adjustment to load changes at data plane speeds raises many interesting challenges, such as computing path weight distribution in a resource-constrained environments like programmable data planes.

Traffic classification. For some workloads, operators may prefer to split traffic over multiple paths, whereas for other workloads splitting may not preferred. Such flexibility could be enabled by having a policy language (*e.g.*, Contra [15]) with appropriate programming abstractions. The underlying system would generate policy-compliant probes over paths based on the desired policy. Also, the system would generate switch-level code that classifies workloads and enforces the policy—*e.g.*, whether or not to split a certain kind of workloads.

Handling out-of-order packet delivery. DASH may split packets of a flow over multiple paths, especially when an update is in progress. This can lead to out-of-order packet delivery. To mitigate out-of-order delivery, DASH can be integrated with *flowlet switching* [5, 18], where packets are grouped into flowlets, and the forwarding decision is updated at the granularity of flowlets. To be specific, the switch remembers the path on which the first packet of a flowlet is sent, and all subsequent packets in the flowlet are sent along this path.

8 CONCLUSION

We have studied the tradeoffs of different novel data structures for load-aware traffic splitting in the data plane. We presented DASH, a data structure where the core idea is to assign a small number of hash boundaries to each stage and maintain the hash boundaries by leveraging multiple pipeline stages and per-stage SALUs. By doing so, DASH enables accurate traffic splitting and efficient updates to the path weights. Compared to the WCMP-based algorithms which take thousands of packets, DASH takes significantly fewer—tens of packets—to update path weights. DASH complements existing solutions that operate *entirely* in the data plane to balance load dynamically across multiple paths.

Acknowledgments: We thank our shepherd Sandesh Kumar Sodhi, anonymous reviewers, Robert MacDavid, Xiaoqi Chen, and Satadal Sengupta for their insightful comments and suggestions. This work was partially supported by CNS-1801884, CNS-1703493, and DARPA DCOMP HR0011-17-C-0047.

REFERENCES

[1] P4-Programmable Ethernet Switch ASIC. https://www.barefootnetworks.com/products/brief-tofino/.
[2] Mpls traffic engineering. https://blog.ipspace.net/2007/02/unequal-cost-load-sharing.html.
[3] Ns-3 simulator. https://www.nsnam.org/.
[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *USENIX NSDI*, 2010.
[5] 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 *ACM SIGCOMM*, 2014.
[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM*, 2010.
[7] C. H. Benet, A. J. Kassler, T. Benson, and G. Pongracz. MP-HULA: Multipath transport aware load balancing using programmable data planes. In *NetCompute*, 2018.
[8] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM*, 2013.
[9] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM*, 2013.
[10] A. Elwalid, C. Jin, S. Low, and I. Widjaja. Mate: MPLS adaptive traffic engineering. In *IEEE INFOCOM*, 2001.
[11] R. Gallager. A minimum delay routing algorithm using distributed computation. *IEEE Transactions on Communications*, 1977.
[12] J. He, M. Suchara, M. Bresler, J. Rexford, and M. Chiang. Rethinking internet traffic management: From multiple decompositions to a practical protocol. In *ACM CoNEXT*, 2007.
[13] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.
[14] C. Hopps. Analysis of an equal-cost multi-path algorithm. https://tools.ietf.org/html/rfc2992, 2000.
[15] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, P. Tammana, and D. Walker. Contra: A programmable system for performance-aware routing, to appear in NSDI, 2020.
[16] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM*, 2013.
[17] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *ACM SIGCOMM*, 2005.
[18] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Dynamic load balancing without packet reordering. *SIGCOMM Comput. Commun. Rev.*
[19] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford. Efficient traffic splitting on commodity switches. In *ACM CoNEXT*, 2015.
[20] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *ACM SOSR*, 2016.
[21] P. Kumar, Y. Yuan, C. Yu, N. Foster, R. Kleinberg, P. Lapukhov, C. L. Lim, and R. Soulé. Semi-oblivious traffic engineering: The road not taken. In *USENIX NSDI*, 2018.
[22] N. Michael and A. Tang. Halo: Hop-by-hop adaptive link-state optimal routing. *IEEE/ACM Transactions on Networking*, 2015.
[23] D. Xu, M. Chiang, and J. Rexford. Link-state routing with hop-by-hop forwarding can achieve optimal traffic engineering. In *IEEE INFOCOM*, 2008.
[24] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat. WCMP: Weighted cost multipathing for improved fairness in data centers. In *EuroSys*, 2014.