Scalable, Optimal Flow Routing in Datacenters via Local Link Balancing

Siddhartha Sen, David Shue, Sunghwan Ihm, and Michael J. Freedman
Princeton University

ABSTRACT

Datacenter networks should support high network utilization. Yet today's routing is typically load agnostic, so large flows can starve other flows if routed through overutilized links. Recent proposals for datacenter routing, such as centralized scheduling or end-host multi-pathing, do not offer optimal throughput, and they suffer from scalability concerns and other limitations.

We observe that most datacenter networks have a symmetry property that admits a better solution. We develop a simple, switch-local algorithm called LocalFlow that routes the maximum multi-commodity flow in these networks, while tolerating failures and asymmetry. LocalFlow evades existing hardness results by allowing flows to be fractionally split, but it minimizes the number of split flows by considering the *aggregate* flow per destination and allowing *slack* in the splitting. Through an optimization decomposition, we show that LocalFlow, in conjunction with unmodifed end hosts' TCP, achieves an optimal solution. Splitting flows presents several new technical challenges that must be overcome in order to achieve high accuracy, interact properly with TCP, and be implementable on emerging standards for programmable, commodity switches.

LocalFlow acts independently on each switch. This makes it highly scalable, allows it to adapt quickly to dynamic workloads, and enables flexibility in the deployment of its control-plane scheduling logic. We present detailed packet-level simulations that demonstrate LocalFlow's practicality and optimality, comparing it to a variety of alternative schemes and configurations, using distributions and traces from real datacenter workloads.

1. INTRODUCTION

The growth of popular Internet services and cloud-based platforms has spurred the construction of large-scale datacenters, which may contain thousands to hundreds of thousands of servers. Traditional enterprise network architectures fare poorly in such environments [2], however, which has led to a rash of research proposals for new datacenter networking architectures. Many such architectures (e.g., [2, 19]) are based on Clos topologies [11]; they primarily focus on increasing *bisection bandwidth*, or the communication capacity between any bisection of the end hosts. Unfortunately, even with full bisection bandwidth, the utilization of the core network suffers when large flows are routed poorly, as collisions with other flows can limit their throughput even while other, less utilized paths, are available (see Figure 2).

The problem of simultaneously routing flows through a capacitated network is the *multi-commodity flow (MCF) problem*. This problem has been studied extensively by both the theoretical and networking systems communities. Solutions deployed in datacenters today are typically load agnostic, however, such as Equal-Cost Multi-Path (ECMP) [21] and Valiant Load Balancing (VLB) [35]. More recently, the networking community has proposed a series of load-aware solutions including both centralized solutions (e.g., [3, 6]), where routing decisions are made by a global scheduler, and distributed solutions, where routing decisions are made by end hosts (e.g., [24, 36]) or switches (e.g., [25]).

As we discuss in §2, these current approaches have limitations. Centralized solutions like Hedera [3] face serious scalability and fault-tolerance challenges with today's datacenter workloads [5, 26]. End-host solutions like MPTCP [36] offer greater parallelism but lack a global view of the network, forcing them to continuously react to path congestion. Switch-local solutions like FLARE [25] scale well, but their local view of network traffic makes efficient routing difficult. In fact, we are unaware of any prior load-aware, switch-local solution that is practical for datacenter networks.

Only a few of these current solutions split flows across multiple paths. This makes the majority of them suboptimal, by the hardness of the unsplittable multi-commodity flow problem [17]. Indeed, our experimental evaluation shows a significant throughput drop when flows are not split. Yet at the same time, splitting flows is problematic in practice because it causes packet reordering, which in the case of TCP may lead to throughput collapse [27]. Solutions that do split flows are either load agnostic [9, 13], use non-trivial protocols [18, 36], or rely on specific traffic patterns [25].

Empowered by the rise of commodity switches with software-controlled dataplanes (e.g., via OpenFlow [29]), but cognizant of their limitations in centralized solutions [12], this paper explores whether purely switch-local algorithms could better handle today's high-scale and dynamic datacenter traffic patterns. We introduce LocalFlow, the first practical switch-local algorithm that routes flows optimally in *symmetric* networks, a property we define later. Most proposed datacenter architectures (e.g., fat-trees [2, 19]) and real deployments satisfy the symmetry property. To derive our solution, we decompose the MCF optimization problem using a dual-dual approach [8, 31], and show that the slave dual component is essentially solved by end hosts' TCP, while the master dual component can be solved locally at each switch by LocalFlow. In fact, a naïve scheme called PacketScatter [9, 13], which essentially round-robins packets over a switch's outgoing links, solves the master component

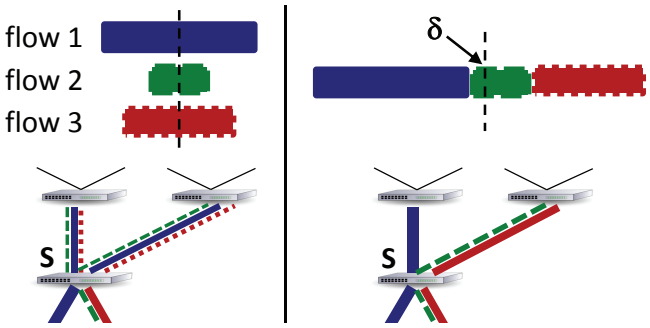


Figure 1: A set of flows to the same destination arrives at switch S. PacketScatter (left) splits each individual flow, whereas LocalFlow (right) distributes the aggregate flow, and only splits an individual flow if the load imbalance exceeds δ.

optimally in symmetric networks. However, PacketScatter is load agnostic: it splits *every* flow, which causes packet reordering and increases flow completion times, and it cannot handle network failures or asymmetry.

LocalFlow overcomes these limitations with the following insights. By considering the *aggregate* flow to each destination, rather than individual transport-level flows, we split at most $|L|-1$ flows, where $|L|$ is the number of candidate outgoing links (< 10 in most cases). By further allowing splitting to be *approximate*, using a slack parameter $\delta \in [0,1]$, we split even fewer flows (or possibly none!). Figure 1 illustrates these ideas. In the limit, setting $\delta = 1$ yields a variant of LocalFlow that schedules flows as indivisible units; we call this variant LocalFlow-NS ("no split"). Like PacketScatter, LocalFlow *proactively* avoids congestion, allowing it to automatically cope with traffic unpredictability, a major headache in other solutions [6, 36]. However, by using flexible, load-aware splitting, LocalFlow splits much fewer flows and can even tolerate failures and asymmetry in the network.

The benefits of a switch-local algorithm are profound. Because it acts locally without coordination between switches, LocalFlow can operate at very small scheduling intervals at an unprecedented scale. This allows it to adapt to highly dynamic traffic patterns. At the same time, LocalFlow admits a wide variety of deployment options of its control-plane logic. For example, the scheduler could run locally on the switch's CPU itself, or more remotely on commodity servers spread throughout the network, or even on a single server for the entire network. In all cases, however, the scheduling performed for each switch is *independent* of that for others. On the flipside, a purely local algorithm has its limitations. For example, it cannot optimally route a dynamic workload when faced with network failures or asymmetry.

Splitting flows introduces several technical challenges in order to achieve high accuracy, use modest forwarding table state, and interact properly with TCP. (Although LocalFlow may be used with UDP, this paper primarily focuses on TCP traffic.) Besides minimizing the number of flows that are split, LocalFlow employs two novel techniques to split flows efficiently. First, it splits flows *spatially* for higher accuracy, by installing carefully crafted rules into switches' forwarding tables that partition a monotonically increasing sequence number. Second, it supports splitting at *multiple resolutions* to control forwarding table expansion, so rules can represent groups of flows, single flows, or subflows. Our mechanisms for implementing multi-resolution splitting use existing (for LocalFlow-NS) or forthcoming (for LocalFlow) features of OpenFlow-enabled switches [1]. Given the forthcoming nature of one of these features, and our desire to evaluate LocalFlow at scale, our evaluation focuses on simulations. We use a network simulator that implements full packet-level TCP behavior [36], as well as simulations based on real datacenter traces [5, 19].

Our evaluation shows that LocalFlow achieves near-optimal throughput, outperforming ECMP by up to 171%, MPTCP by up to 19%, and Hedera by up to 23%. Compared to PacketScatter which splits all flows over time, it split less than 4.3% of flows on a real switch packet trace and achieved lower flow completion times throughout. By modestly increasing the duplicate-ACK threshold of end hosts' TCP, LocalFlow avoids the adverse effects of packet reordering. Interestingly, the high accuracy of spatial splitting turns out to be crucial, as even slight load imbalances, such as those incurred by temporal splitting (e.g., [25]), significantly degrade throughput (e.g., by 17%). Our evaluation also uncovered several other interesting findings, such as the high throughput of LocalFlow-NS on VL2 topologies [19].

We next compare LocalFlow to the landscape of existing solutions. We define our network architecture as well as the symmetry property in §3. We present the LocalFlow algorithm in §4 and our multi-resolution splitting technique in §5. We conduct a theoretical analysis of LocalFlow in §6, evaluate it in §7, and then conclude.

2. EXISTING APPROACHES

We discuss a broad sample of existing flow routing solutions along two important axes, scalability and optimality, while comparing them to LocalFlow. Scalability encompasses a variety of metrics, including forwarding table state at switches, network communication, and scheduling frequency. Optimality refers to the maximum flow rate achieved relative to optimal routing.

Centralized solutions typically run a sequential algorithm at a single server [3, 6, 7]. These solutions lack scalability, because the cost of collecting flow information, computing flow paths, and deploying these paths makes it impractical to respond to dynamic workloads. Indeed, coordinating decisions in the face of traffic burstiness and unpredictability is a major problem [6], whereas LocalFlow avoids such coordination. The rise of switches with externally-managed forwarding tables, such as OpenFlow [1, 29], has increased the responsiveness of centralized solutions. For example, Hedera's scheduler runs every 5 seconds, with the potential to run at subsecond intervals [3], and MicroTE schedules flows every second [6]. But recent studies [5, 26] have concluded that the size and workloads of today's datacenters require *parallel* route setup on the order of *milliseconds*, making a centralized OpenFlow solution infeasible even in small datacenters [12]. This infeasibility motivated our pursuit of a parallel solution.

End-host solutions employ more parallelism, and most give provable guarantees. TeXCP [24] and TRUMP [20] dynamically load-balance traffic over multiple paths between pairs of ingress-egress routers (e.g., MPLS tunnels) established by an underlying routing architecture. DARD [37] is a similar solution for datacenter networks that controls paths from end hosts. (We discuss MPTCP [36] further below.) These solutions explicitly model paths in their formulation, leading to exponential representation and convergence time in the worst case. To cope with this, all solutions use only a handful of paths per source-destination pair. Since end-host solutions lack information about other flows in the network, they must continuously react to congestion on paths and rebalance load.

Switch-local solutions have more visibility of active flows, especially at aggregation and core switches, but still lack a global view of the network. They achieve high scalability and can evade the exponential representation problem. For example, REPLEX [14] gathers (aggregate) path information using measurements on adjacent links and by exchanging messages between neighbors.

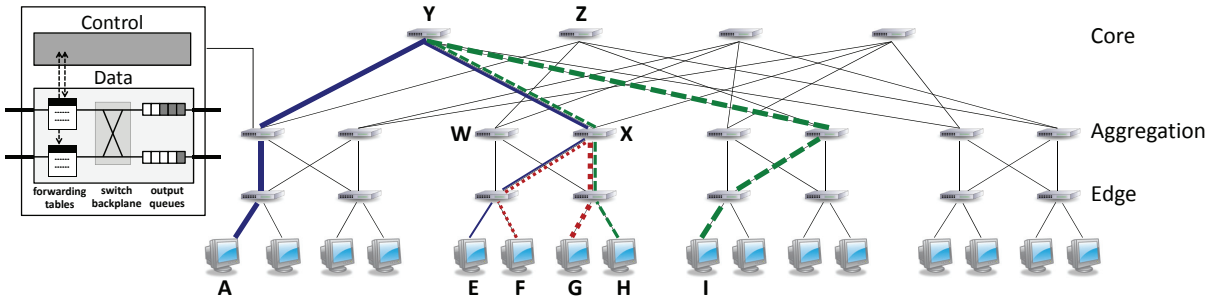


Figure 2: A FatTree network with 4-port switches. VL2 is a variation on this topology. End hosts A, G, I simultaneously transmit to E, F, H and collide at switches Y and X, but there is sufficient capacity to route all flows at full rate.

None of the above solutions split individual flows, however, and hence cannot produce optimal results, since the unsplittable multi-commodity flow problem is NP-hard and admits no constant-factor approximation [17]. MPTCP [32, 36] is an end-host solution that splits a flow into subflows and balances load across subflows via linked congestion control. It uses two levels of sequence numbers and buffering to handle reordering across subflows. DeTail [38] modifies switches to do per-packet adaptive load balancing based on queue occupancy, with the goal of reducing the flow completion time tail. It relies on layer-2 backpressure and modifications to end hosts' TCP to avoid congestion, as well as additional buffers at the end hosts to handle reordering. Geoffray and Hoefler [18] propose an adaptive source-routing scheme that uses layer-2 back-pressure and probes to evaluate alternative paths. Like DeTail, their scheme requires modifications to both end hosts and switches. FLARE [25] is a general technique for splitting flows that can be combined with systems like TexCP. It exploits delays between packet bursts in wide-area networks to route each burst along a different path without incurring reordering.

It is instructive to compare our solution, LocalFlow, to the above schemes. Like all of them, LocalFlow splits flows, but whereas the above schemes tend to split *every* flow, LocalFlow minimizes the number of flows that are split. This is largely due to the fact that LocalFlow balances load *proactively*, giving it full control over how flows are scheduled and split, instead of reacting to congestion after-the-fact. Unlike most of the above schemes, LocalFlow is purely switch-local and does not modify end hosts. In this respect it is similar to FLARE, but it also differs from FLARE because it splits flows *spatially* (e.g., based on TCP sequence numbers), not temporally. Our simulations show that spatial splitting significantly outperforms temporal splitting (§7.6). Finally, LocalFlow achieves near-optimal routing in practice, which the other solutions have not demonstrated, despite being considerably simpler than all of them.

The only flow-splitting solution that is truly simpler than LocalFlow is the PacketScatter scheme [9, 13] we discussed earlier. LocalFlow can be viewed as a load-aware, efficient version of PacketScatter. We discuss the differences between the two schemes in more detail while deriving LocalFlow's design.

A long history of theoretical algorithms exist for multi-commodity flow that are provably optimal (see, e.g., [16]). However, these solutions are disconnected from practice for reasons we have outlined before [33]. In contrast, LocalFlow has near-optimal performance in both theory and practice.

3. ARCHITECTURE

LocalFlow is a flow routing algorithm designed for datacenter networks. In this section, we describe the components that comprise this network (§3.1), outline the main control loop of a LocalFlow scheduler (§3.2), and define the networks on which LocalFlow is efficient (§3.3).

3.1 Components and deployment

Figure 2 shows a typical datacenter network of end hosts and switches on which LocalFlow might be deployed. The techniques we use are compatible with existing or forthcoming features of extensible switches, such as OpenFlow [1, 29], Juniper NOS [23], or Cisco AXP [10].

The architecture and capabilities of hardware switches differ significantly from that of end hosts, making even simple tasks challenging to implement efficiently. The detail in Figure 2 shows a typical switch architecture, which consists of a control plane and a data plane. The data plane hardware has multiple physical (e.g., Ethernet) ports, each with its own line card, which (when simplified) consists of an incoming lookup/forwarding table in fast memory (SRAM or TCAM) and an outgoing queue. The control plane performs general-purpose processing and can install or modify rules in the data plane's forwarding tables, as well as obtain statistics such as the number of bytes that matched a rule. To handle high data rates, the vast majority of traffic must be processed by the data-plane hardware, bypassing the control plane entirely.

The hardware capabilities, programmability, and processing/memory resources of switches are continually increasing [30]. Being a local algorithm, LocalFlow's demands on these resources are limited to its local view of network traffic, which is orders of magnitude smaller than that of a centralized controller [5, 12, 26]. Nevertheless, since LocalFlow runs independently for each switch, it supports a wide variety of deployment options of its control plane logic. For example, it can run locally on each switch, using a rack-local controller with OpenFlow switches or a separate blade in the switch chassis of Juniper NOS or Cisco AXP switches. Alternatively, the number of these controllers can be reduced, causing them to be shared by switches, and their locations can be changed to meet the scalability requirements of the network. In the limit, a single centralized controller may be used. Note that regardless of the deployment strategy, LocalFlow's independence allows each scheduler to execute on separate threads, processes, cores, or devices.

3.2 Main LocalFlow control loop

Each LocalFlow controller runs a continuous scheduling loop. At the beginning of every interval, it executes the following:

1) Measure the rate of each active flow. This is done by querying the byte counter of each forwarding rule from the previous interval and dividing by the interval length.

2) Run the scheduling algorithm using the flow rates from step 1 as input.

3) Update the rules in the forwarding table based on the outcome of step 2, and reset all byte counters.

Steps 2 and 3 are described in §4 and §5, respectively.

3.3 Symmetric networks

Although LocalFlow can be run on any network, it only achieves optimal throughput on networks that satisfy a certain symmetry property. This property is defined as follows:

DEFINITION 1. *A network is* symmetric *if for all source-destination pairs (s,d), the following holds: all switches p on the shortest paths between s and d that are the same distance from s have identical outgoing capacity to d.*

In other words, any of the p switches are equally good intermediate candidates for routing a flow between s and d. Using the example of Figure 2, switches Y and Z are both on a shortest path between (A,E), and both have one link of outgoing capacity to E.

Real deployments and most proposed datacenter architectures satisfy the symmetry property. For example, it is satisfied by fat-tree-like networks (e.g., [2, 19]), which are Clos [11] topologies arranged as multi-rooted trees that support full-bisection bandwidth. FatTree [2] is a three-stage fat-tree network built using identical k-port switches arranged into three levels—edge, aggregation, and core—supporting a total of $k^3/4$ end hosts. Figure 2 shows a 16-host FatTree network ($k = 4$). VL2 [19] modifies FatTree by using higher-capacity links between Top-of-Rack (ToR, i.e., edge), aggregation, and intermediate (i.e., core) switches. Unlike FatTree, the aggregation and intermediate switches form a complete bipartite graph in VL2.

Real datacenters are typically oversubscribed due to cost and practicality reasons, because it is uncommon for all hosts to simultaneously transmit at full rate (i.e., utilize the full bisection bandwidth). The networks above can be oversubscribed by simply connecting more hosts to each edge/ToR switch, which preserve their symmetry property.

4. ALGORITHM LOCALFLOW

This section presents LocalFlow, our switch-local algorithm for routing flows in symmetric datacenter networks. It is invoked in Step 2 of the main control loop (§3.2). At a high-level, LocalFlow attempts to find the optimal flow routing for the following max multi-commodity flow (MCF) problem:

$$\textbf{maximize: } \sum_i U_i(x_i) \tag{1}$$

$$\textbf{subject to: } \sum_{u:(u,v)\in E} f^{s,d}_{u,v} = \sum_{w:(v,w)\in E} f^{s,d}_{v,w} : \forall v,s,d,$$

$$\sum_{u:(s,u)\in E} f^{s,d}_{s,u} = \sum_{i:s\to d} x_i : \forall s,d$$

$$\sum_{s,d} f^{s,d}_{u,v} \le c_{u,v} : \forall (u,v)\in E, \text{ link capacity } c_{u,v}$$

variables: x_i commodity send rates, $f^{s,d}_{u,v}$ link flow rates

It does so by balancing the flow across links between adjacent switches; this technique is similar to, but more aggressive than, that of Awerbuch and Leighton [4]. The key insight is that if we split a flow evenly over equal-cost links along its path to a destination, then even if it collides with other flows midway, the colliding subflows will be small enough to still route using the available capacity.

Link balancing on its own does not guarantee an optimal solution to the max MCF objective (1), which depends on the per-commodity utility functions U_i. Fortunately, LocalFlow can rely on the end hosts' TCP congestion control for this purpose [28]. By balancing the per-link flow rates, LocalFlow adjusts the flow routing in response to TCP's optimized send-rates, while TCP in turn adapts to the new routing. We show how this interaction achieves the MCF optimum in §6. Intuitively, this works because the splitting strategy above guarantees that all avoidable congestion has been avoided; any congestion beyond this implies that the destination link is oversaturated, and end hosts must throttle their flows.

We first describe a basic load-agnostic solution for link balancing called PacketScatter (§4.1). We then improve this solution to yield LocalFlow (§4.2). Finally, we discuss a simple extension to LocalFlow that copes with failures in the network (§4.3).

4.1 Basic solution: PacketScatter

The simplest solution for link balancing is to split *every* flow over *every* equal-cost outgoing link of a switch ($f^{s,d}_{u,v} = f^{s,d}_{u,w}$). We call this scheme PacketScatter. PacketScatter round-robins packets to a given destination over the switch's outgoing links and has been supported by Cisco switches for over a decade now [9]. Recent work by Dixit et al. [13] studies variants of the scheme that select a random outgoing link for each packet to reduce state. However, this approach is problematic because even slight load imbalances due to randomness can significantly degrade throughput, as our evaluation confirms (§7.6).

While PacketScatter may be optimal in terms of flow routing, because it unconditionally splits *every* flow at individual-packet boundaries, as shown in Figure 1, it can cause excessive reordering at end hosts. These out-of-order packets can inadvertently trigger TCP fast-retransmit, disrupting throughput, or delay the completion of short-lived flows, increasing latency. On the upside, because the splitting is load agnostic, it is highly accurate and oblivious to traffic bursts and unpredictability. However, by the same token, it cannot adapt to partial network failures since it will continue to send flow down under-capacitated subtrees.

4.2 LocalFlow

We obtain LocalFlow by applying three ideas to PacketScatter that remove its weaknesses while retaining its strengths. The pseudocode is given in Algorithm 1.

First, instead of unconditionally splitting every flow, we group the flows by destination d first and then distribute their *aggregate* flow rate evenly over $|L_d|$ outgoing links (lines 2-6 of Algorithm 1). This corresponds to a simple variable substitution : $f^d_{u,v} = \sum_s f^{s,d}_{u,v}$ in the MCF formulation which gives the modified constraints:

$$\sum_{w:(v,w)\in E} f^d_{v,w} - \sum_{u:(u,v)\in E} f^d_{u,v} = \sum_{i:v,d} x_i : \forall v,u\neq d \tag{2}$$

$$\sum_d f^d_{u,v} \le c_{u,v} : \forall (u,v)\in E$$

This means that LocalFlow splits at most $|L_d|-1$ times per destination. Function BINPACK does the actual splitting. It sorts the flows according to some *policy* e.g., increasing rate and successively places them into $|L_d|$ equal-sized bins (lines 17-25). If a flow does not fit into the current least loaded bin, BINPACK splits the flow (possibly unevenly) into two subflows, one which fills the bin and the other which rejoins the sorted list of flows (lines 20-21).

Our second idea is to allow some *slack* in the splitting. Namely, we allow the $|L_d|$ bins to differ by at most a fraction $\delta \in (0,1]$ of the link bandwidth (line 19). This not only reduces the number of flows that are split, but it also ensures that small flows of rate $\le \delta$ are never split. Note that small flows are still bin-packed by the algorithm; it is only the *last* such flow entering a bin that may give rise to an imbalance. After BINPACK returns, LOCALFLOW ensures

```
1  function LocalFlow(flows F, links L)
2    dests D = {f.dest | f ∈ F}
3    foreach d ∈ D do
4      flows F_d = {f ∈ F | f.dest = d}
5      links L_d = {l ∈ L | l is on a path to d}
6      bins B_d = BinPack(F_d, |L_d|)
7      Sort B_d by increasing total rate
8      Sort L_d by decreasing total rate
9      foreach b ∈ B_d, l ∈ L_d do
10       Insert all flows in b into l
11   end

12 bins function BinPack(flows F_d, |links L_d|)
13   δ = ...; policy = ...
14   binCap = (Σ_{f∈F_d} f.rate)/|L_d|
15   bins B_d = {|L_d| bins of capacity binCap}
16   Sort F_d by policy
17   foreach f ∈ F_d do
18     b = argmax_{b∈B_d} b.residual
19     if f.rate > b.residual + δ then
20       {f_1, f_2} = Split(f, b.residual, f.rate − b.residual)
21       Insert f_1 into b; Add f_2 to F_d by policy
22     else
23       Insert f into b
24     end
25   end
26   return B_d
```

Algorithm 1: Our switch-local algorithm for routing flows on fat-tree-like networks.

that larger bins are placed into less loaded links (lines 7-10). This ensures that the links stay balanced to within δ even after all destinations have been processed, as we prove in Lemma 6.2. Figure 1 illustrates the above two ideas. In the example shown, no flows are actually split by LocalFlow because they are accommodated by the δ slack while PacketScatter splits every flow.

Since LocalFlow may split only a few flows, and may split a flow over a subset of the outgoing links, possibly unevenly, we cannot use the (load-agnostic) round-robin splitting scheme of PacketScatter. Instead, we introduce a new, load-aware scheme called *multi-resolution splitting* that splits traffic in a flexible manner, by installing carefully crafted rules into the forwarding tables of switches. Along with their current rates (as measured by Step 1 of the main control loop), these rules comprise the set F that is input to function LocalFlow. Multi-resolution splitting (the implementation of Split) is discussed in §5.

These changes eliminate the weaknesses of PacketScatter while retaining its strengths. Even though LocalFlow's splitting strategy is load aware, it is still based on purely local measurements and actions, which allows it to cope with traffic burstiness and unpredictability. By using flow and subflow rules, LocalFlow retains the accuracy of round-robin splitting at the cost of increased state (see §5.3).

4.3 Handling failures and asymmetry

Perhaps surprisingly, many failures in a symmetric network can be handled seamlessly, because they do not violate the symmetry property. In particular, complete node failures—that is, failed end hosts or switches—remove all shortest paths between a source-destination pair that pass through the failed node. For example, if switch X in Figure 2 fails, all edge switches in the pod now have only one option for outgoing traffic: switch W. The network is still symmetric, so LocalFlow's optimality (Theorem 6.3) still holds. Indeed, even PacketScatter can cope with such failures.

Partial failures—that is, individual link or port failures, including slow (down-rated) links—are more difficult to handle, because they violate the symmetry property. For example, consider when switch X in Figure 1 loses one of its uplinks. PacketScatter at the edge switches would continue to distribute traffic equally between switches W and X, even though X has half the outgoing capacity as W. This results in suboptimal throughput. However, with a simple modification, LocalFlow is able to cope with this type of failure. When switch X experiences the partial failure, other LocalFlow controllers learn about it from the underlying link-state protocol (which automatically disseminates this connectivity information), or by piggy-backing over the underyling distance-vector protocol. The upstream controllers determine the fraction of capacity lost and use this information to rebalance traffic sent to W and X, by simply modifying the bin sizes used in lines 14-15 of Algorithm 1. In this case, LocalFlow sends twice as much traffic to W than X. Note that it may not be possible to send this much traffic to W (due to link capacity constraints), and it may even be the case that distributing traffic equally would have fared better (e.g., if there are no competing flows in the network). In general, determining the optimal rebalancing strategy requires non-local knowledge of the current flows. This is the cost of asymmetry.

Another advantage of LocalFlow in the above situation is that, because it splits fewer flows than PacketScatter, fewer flows are likely to be affected by the link failure. This can be seen in Figure 1.

Since failures in a symmetric network effectively introduce asymmetry, a scheme similar to the one above can be used to run LocalFlow in an asymmetric network. However, as we noted, determining the optimal link weights to use at each switch may require non-local knowledge of the current traffic, which LocalFlow does not have access to. disposal. We believe that simple heuristics that should do reasonably well, however.

5. MULTI-RESOLUTION SPLITTING

Multi-resolution splitting is our spatial splitting technique for implementing the Split function in Algorithm 1. It splits traffic at different granularities by installing carefully crafted rules into the forwarding tables of emerging programmable switches [1, 34]. Figure 3 illustrates each type of rule. These rules can represent single flows and subflows (§5.1), leading to a variant of LocalFlow that is very accurate and time-efficient. They can also represent "metaflows" (§5.2), or groups of flows to the same destination, leading to a variant of LocalFlow that is less accurate but more space-efficient.[1] We describe the two LocalFlow variants in §5.3.

5.1 Flows and subflows

To represent a single flow, we install a forwarding rule that exactly specifies all fields of the flow's 5-tuple. This uniquely identifies the flow and thus matches all of its packets.

To split a flow into two or more subflows, we use one of two techniques. The first technique extends a single-flow rule to additionally match bits in the packet header that *change* during the flow's lifetime, e.g., the TCP sequence number. (Currently, OpenFlow switches do not support matching bits of the TCP sequence number, although its roadmap [34] suggests that functionality will appear in an upcoming version.) To facilitate finer splitting at later switches, we group packets into contiguous blocks of at least t bytes, called *flowlets*, and split only along flowlet boundaries. Our notion of

[1] Since metaflow and subflow rules use partial wildcard matching, they must appear in the TCAM of the switch, a scarcer resource. However, our simulations show that LocalFlow minimizes the amount of splitting and hence the number of rules required.

Type	Src IP	Src Port	Dst IP	Dst Port	TCP seq/counter	Link
M	*	*11	E	*	*	1
	*	*10	E	*	*	2
	*	*	E	*	*	3
F	A	u	F	v	*	1
S	A	x	G	y	*0**********	1
	A	x	G	y	*10**********	2
	A	x	G	y	*11**********	3

Figure 3: Multi-resolution splitting rules (M = metaflow, F = flow, S = subflow).

flowlets is *spatial* and thus different from that of FLARE [25], which crucially relies on timing properties.

By carefully choosing which bits to match and the number of rules to insert, we can split flows with different ratios and flowlet sizes. Specifically, to split a flow evenly over L links with flowlet size t, we add L forwarding rules for each possible $\lg L$-bit string whose least significant bit starts after bit $\lceil \lg t \rceil$ in the TCP sequence number.[2] Since TCP sequence numbers increase consistently and monotonically, this causes the flow to match a different rule every t bytes. Also, since initial sequence numbers are randomized, the flowlets of different flows are also desynchronized. Uneven splitting can be achieved in a similar way. For example, the subflow rules in Figure 3 split a single flow over three links with ratios $(1/4, 1/4, 1/2)$ and $t = 1024$ bytes. By using more rules, we can support uneven splits of increasing precision.

Since later switches along a path may need to further split subflows from earlier switches, they should use a smaller flowlet size than the earlier switches. In hierarchical networks like fat-trees, this means the flowlet size should decrease as you go up the tree. For example, edge switches in Figure 2 may use $t = 2$ maximum segment sizes (MSS) while aggregation switches use $t = 1$ MSS. In general, smaller flowlet sizes lead to more accurate load balancing, as our evaluation confirms.

An alternative technique that avoids the need for flowlets is to associate a counter of bytes with each flow that is split, and increment it whenever a packet from that flow is sent. Such counters are common in OpenFlow switches [1]. The value of the counter is used in place of the TCP sequence number in the subflow rules of Figure 3. Since each switch uses its own counter to measure each flow, we no longer rely on contiguous bytes (flowlets) for downstream switches to make accurate splitting decisions. The counter method is also appropriate for UDP flows, which do not have a sequence number in their packet headers.

Compared to the above techniques, temporal splitting techniques like FLARE are inherently less precise, because they rely on unpredictable timing characteristics such as delays between packet bursts. For example, during a bulk data shuffle between MapReduce stages, there may be few if any intra-flow delays. This lack of precision leads to load imbalances that significantly degrade throughput, as shown in §7.6.

5.2 Metaflows

To represent a metaflow, we install a rule that specifies the destination IP field but uses wildcards to match all other fields. This matches all flows to the same target, illustrated by the third metaflow rule in Figure 3. To split a metaflow, we additionally specify some least significant bits (LSBs) in the source port field. In the example, the metaflow rules split all flows to target E over three links with ratios $(1/4, 1/4, 1/2)$. Note that the "all" rule is placed on the bottom to illustrate its lower priority (although, unlike here, priorities are explicit in OpenFlow); it captures the remaining $1/2$ ratio.

If source ports are diverse, this scheme splits the *number* of flows by the desired ratios. It may not split the total flow rate by these ratios, however, as individual flow rates may vary. We describe how to use metaflow rules effectively despite such inaccuracies below.

5.3 LocalFlow variants

Using the above techniques, we obtain a straightforward implementation of LocalFlow based on flow and subflow rules. When the first packet of a flow arrives at a switch, the control plane is notified and subsequently installs a single-flow rule for the flow. After every scheduling interval, LocalFlow may decide to split one or more flows. Since each forwarding rule represents one flow (in whole or in part), and splitting is at precise, spatial boundaries (per §5.1), LocalFlow achieves very accurate splitting. On the downside, it requires one or more rules per flow, more than the per-destination rules of PacketScatter.

If we forgo tracking individual flows, we can obtain a variant of LocalFlow that is both more space-efficient and requires fewer reactive rule installations by the control plane. Now, the control plane becomes involved only when the switch encounters a flow to a previously unseen destination, at which time it creates a *metaflow* rule that matches all flows to the destination. Then, we use the technique described in §5.2 to split this metaflow.

Since metaflow splitting does not split individual flows, this scheme may not achieve the desired load balance, even after several scheduling intervals. To cope with this, if the same metaflow rule has been split more than $O(\log |F|)$ times, where $|F|$ is an estimate of the number of active flows, we use subflow splitting during the next split, which is guaranteed to be accurate. The motivation for $O(\log |F|)$ is the bad scenario where many active flows match the metaflow rule, but only one is large (i.e., a single "elephant" among the "mice"). Since metaflow splitting eliminates a constant fraction of the *number* of flows in each split, after $O(\log |F|)$ splits, the large flow will be isolated. Jose et al. [22] used a similar technique to identify large traffic sources in wildcard flow rules.

Our evaluation of LocalFlow suggests that it splits very few flows in practice (§7.4). Thus, we use the first variant in our evaluation, as it is simpler, faster, and more accurate.

6. ANALYSIS

We begin by analyzing the local complexity of LocalFlow, and then move on to our main analysis, which proves its optimality.

During each round, a LocalFlow controller executes $O(|F| \log |F| + \sum_d |F_d| \log |F_d|) = O(|F| \log |F|)$ sequential steps if $\delta = 0$, since it need not sort the bins and links in lines 7-8, where $|F_d|$ is the number of flows to destination d. If $\delta > 0$, the controller executes $O(|F| \log |F| + |F||L| \log |L|)$ steps, where $|L|$ is the number of outgoing links. Relative to the number of active flows, $|L|$ can be viewed as a constant. In terms of space complexity, the first variant of LocalFlow described in §5.3 maintains at least one rule per flow, while the second variant maintains at least one rule per destination. Both of these numbers increase when flow rules are split, in a manner that depends on the workload. We measure LocalFlow's space overhead on a real datacenter workload in §7.4.

We now show that, in conjunction with TCP, LocalFlow maximizes the total network utility (1). Since LocalFlow and TCP alternately optimize their respective variables, we first show that the "master" LocalFlow optimization adapts link flow rates $f^t_{u,v}$ to min-

[2] Since TCP sequence numbers represent a byte offset, the bit string should actually start after bit $\lceil \lg(t \times M) \rceil$, where M is the maximum number of bytes in a TCP segment.

imize the max link cost (i.e., utilization $\frac{\sum_t f^t_{u,v}}{c_{u,v}}$) for commodity flow rates x^*_i determined by the "slave" TCP sub-problem. Then we examine the optimality conditions for TCP and show how the "link-balanced" LocalFlow flow rates lead to an optimal solution to the original MCF objective.

LEMMA 6.1. *If $\delta = 0$, algorithm LocalFlow routes the min cost MCF with fixed commodity flow rates.*

PROOF SKETCH. The symmetry property from §3.3 implies that all outgoing links to a destination lead to equally-capacitated paths. Thus, the maximum load on any link is minimized by splitting a flow equally over all outgoing links; this is achieved by lines 6-10 of LocalFlow. No paths longer than the shortest paths are used, as they necessarily must intersect with a shortest path and thus add to that path's load.

Since we can view multiple flows with the same destination as a single flow originating at the current switch, grouping does not affect the distribution of flow. Repeating this argument for each destination independently yields the min-cost flow. □

When $\delta > 0$, LocalFlow splits the total rate to a destination d over $|L|$ outgoing links, such that no link receives more than δ the flow rate of another. This process is repeated for all $d \in D$ using the same set of links L. Then,

LEMMA 6.2. *At the end of LocalFlow, the total rate per link is within an additive δ of each other.*

PROOF. The lemma trivially holds when $|L| = 1$ because no splitting occurs. Otherwise, the proof is an induction over the destinations in D. Initially there are no flows assigned to links, so the lemma holds. Suppose it holds prior to processing a new destination. Let the total rate on the bins returned by BINPACK be $y_1, y_2, \ldots, y_L$ in increasing order; let the total rate on the links be $x_1, x_2, \ldots, x_L$ in decreasing order. After line 10, the total rate on the links is $x_1 + y_1, x_2 + y_2, \ldots, x_L + y_L$. If $1 \le i < j \le L$ are the links with maximum and minimum rate, respectively, then we have $(x_i + y_i) - (x_j + y_j) = (x_i - x_j) + (y_i - y_j) \le \delta$, since $y_i \le y_j$ and $x_i - x_j \le \delta$ by the inductive hypothesis. The case when $j < i$ is similar. □

Lemma 6.2 does not prevent δ-approximate splits from different switches combining unfavorably later on. However, this scenario is unlikely to occur in practice, and even less likely to persist.

THEOREM 6.3. *LocalFlow, in conjunction with end-host TCP, achieves the max MCF optimum.*

PROOF. To show that the LocalFlow's "link-balanced" flow rates enable TCP to maximize the true max MCF objective (1), we turn to the node-centric NUM formulation [8] of the TCP sub-problem, adapted for the multi-path setting which allows flow splits.

$$\begin{aligned}
\textbf{maximize: } & \sum_i U_i(x_i) \\
& \sum_i x_i \sum_{p:(u,v)\in p} \pi^p_i \le c_{u,v}\text{: } \forall (u,v) \in E \\
& \sum_p \pi^p_i = 1\text{: } \forall i \\
\textbf{variables: } & x_i \ge 0
\end{aligned}$$

Here, LocalFlow has already computed the set of flow variables $f^t_{u,v}$ which have been absorbed into the path probabilities π^p_i. Each π^p_i determines the proportion of commodity x_i's traffic that traverses path p, where p is a set of links connecting source s to destination t. These variables are derived from the link flow rates, $\pi^p_i = \prod_{(u,v)\in p} \frac{f^t_{u,v}}{\sum_{w:(u,w)\in E} f^t_{u,w}}$, and thus implicitly satisfy the original MCF flow and demand constraints (2).

To examine the effect of LocalFlow on the MCF objective, we focus on the optimality conditions for TCP which solves the optimization using dual decomposition [8]. In this approach, we first form the Lagrangian $L(x, \lambda)$ by introducing dual variables $\lambda_{u,v}$, one for each constraint.

$$L(x,\lambda) = \sum_i \int f(x_i)\,\mathrm{d}x_i - \sum_{u,v} \lambda_{u,v} \left(\sum_i x_i \sum_{p:(u,v)\in p} \pi^p_i - c_{u,v} \right)$$

For generality, we define the TCP utility to be a concave function where $U_i(x_i) = \int f(x_i)\,\mathrm{d}x_i$, as in [28], and f is differentiable and invertible. Most TCP utilities fall in this category. Next, we construct the Lagrange dual function $Q(\lambda)$ maximized with respect to x_i:

$$x^*_i = f^{-1}(\beta) \text{ when } \frac{\partial L}{\partial x_i} = 0\ \forall x_i,\ \beta = \sum_p \pi^p_i \sum_{(u,v)\in p} \lambda_{u,v} \quad (3)$$

$$Q(\lambda) = \sum_i \left(\int f(x^*_i)\,\mathrm{d}x^*_i - f^{-1}(\beta)\beta \right) + \sum_{u,v} \lambda_{u,v} \cdot c_{u,v}$$

Minimizing Q with respect to λ gives both the optimal dual and primal variables, since the original objective is concave.

$$\sum_i f^{-1}(\beta) \sum_{p:(u,v)\in p} \pi^p_i = c_{u,v} \text{ when } \frac{\partial Q}{\lambda_{u,v}} = 0, \quad \forall (u,v) \in E \quad (4)$$

When (4) is satisfied, the system has reached the maximum network utility (1). TCP computes this solution in a distributed fashion using gradient descent. End-hosts adjust their local send rates x_i according to implicit measurements of path congestion $\sum_{(u,v)\in p} \lambda_{u,v}$ and switches update their per-link congestion prices $\lambda_{u,v}$ (queuing delay) according to the degree of backlog.

According to the symmetry property, all nodes at the same distance from source s along the shortest paths must have links of equal capacity to nodes in the next level of the path tree. Thus, for all links from a node u to nodes (v, w, etc) in the next level of a path tree, for any source-destination pair, we have:

$$\sum_i f^{-1}(\beta) \sum_{p:(u,v)\in p} \pi^p_i = \sum_i f^{-1}(\beta) \sum_{p:(u,w)\in p} \pi^p_i \quad (5)$$

We know that the set of commodities i that traverse these links are the same, since they are at the same level in the path tree. Thus, we can satisfy (4) by ensuring that the per-commodity values of (5) are equal $\forall i$. PacketScatter satisfies this trivially by splitting every commodity evenly across the equal-cost links ($f^{s,d}_{u,v} = f^{s,d}_{u,w}$):

$$x^*_i \sum_{p:(u,v)\in p} \pi^p_i = x^*_i \sum_{p:(u,w)\in p} \pi^p_i, \quad \forall i$$

Recall that the optimal $x^*_i = f^{-1}(\beta)$, thus satisfying (5)

LocalFlow, on the other hand, groups commodities by destination when balancing flow rate across links and only splits individual commodities when necessary. However, by the same argument for commodities, we know that the set of destinations traversing the links will be the same as well. Thus, if we group the commodities in (4) by destination d then the condition is satisfied when:

$$\sum_{i:s\to d} f^{-1}(\beta) \sum_{p:(u,v)\in p} \pi^p_i = \sum_{i:s\to d} f^{-1}(\beta) \sum_{p:(u,w)\in p} \pi^p_i, \quad \forall t$$

Since LocalFlow distributes per-destination flow evenly across equal-

cost links, $f^d_{u,v} = f^d_{u,w} \forall d$, we have:

$$\sum_{i:s\to d} x_i^* \sum_{p:(u,v)\in p} \pi_i^p = \sum_{i:s'\to d} x_i^* \sum_{p:(u,w)\in p} \pi_i^p \quad (6)$$

By substituting in (3), we satisfy the per-destination optimality condition for (4). Note that LocalFlow will continue to adjust flow rates to achieve (6) in response to TCP's optimized send rates (and vice-versa). Since LocalFlow minimizes the max link utilization by balancing per-destination link flow rates, it opens up additional head room on each link for the current commodity demand x_i^* to grow. At the faster timescale TCP maximizes its demand objective to consume the additional capacity. The process continues until the network converges to an optimal utility (1). □

7. EVALUATION

In this section, we evaluate LocalFlow to demonstrate its practicality and to justify our theoretical claims. Specifically, we answer the following questions:

- Does LocalFlow achieve optimal throughput? How does it compare to Hedera, MPTCP, and other schemes? (§7.2)
- Does LocalFlow tolerate network failures? (§7.3)
- Given the potential for larger rule sets, how much forwarding table space does LocalFlow use? (§7.4)
- Do smaller scheduling intervals give LocalFlow an advantage over centralized solutions (e.g., Hedera)? (§7.5)
- Is spatial flow splitting better than temporal splitting (e.g., as used by FLARE)? (§7.6)
- How well does LocalFlow manage packet reordering compared to PacketScatter, and what is its effect on flow completion time? (§7.7)
- When are end-host solutions (e.g., MPTCP) preferable to LocalFlow? (§7.8)

We use different techniques to evaluate LocalFlow's performance, including analysis (§6, §7.8) and simulations on real datacenter traffic (§7.4), but the bulk of our evaluation (§7.2-§7.7) uses a packet-level network simulator. Packet-level simulations allow us to isolate the causes of potentially complex behavior between LocalFlow and TCP (e.g., due to flow splitting), to test pessimistic scenarios that are difficult to construct in practice, and to facilitate comparison with prior work. In fact, we used the same simulator codebase as MPTCP [32, 36], allowing direct comparisons.

7.1 Implementation and experimental setup

Simulations. We developed two simulators for LocalFlow. The first is a stand-alone simulator that runs LocalFlow in isolation on `pcap` packet traces. We used the packet traces collected by Benson et al. [5] from a university datacenter switch. To stress the algorithm, we simulated the effect of larger flows by constraining the link bandwidth of the switch.

Our second simulator is based on *htsim*, a full packet-level network simulator written by Raiciu et al. [32, 36]. The simulator models TCP and MPTCP in similar detail to ns2, but is optimized for larger scale and high speed. It includes an implementation of Hedera's First Fit heuristic [3]. We modified and extended *htsim* to implement the LocalFlow algorithm.

The *htsim* simulator models arbitrary networks using *pipes* (that add delays) and *queues* (with fixed processing capacity and finite buffers). We added a *switch* abstraction that groups queues together and maintains a forwarding table for routing. The forwarding table supports the multi-resolution splitting rules defined in §5; these rules are manipulated by LocalFlow every scheduling interval, independently for each switch. For subflow rules, we used the flowlet technique instead of counters. We did not use metaflow rules in our experiments, partly encouraged by the results in §7.4, which show that LocalFlow's space utilization is modest.

We allowed the duplicate-ACK (dup-ACK) threshold of end-host TCP to be modified (the default is 3), but otherwise left end hosts unchanged. Changing the threshold is easy in practice: on Linux, for example, one simply writes to the `proc` filesystem at `/proc/sys/net/ipv4/tcp_reordering`.

Topologies. We ran our experiments on the different fat-tree-like topologies described in §3, including:

- FatTree topology built from k-port switches [2]. We used 1024 hosts ($k = 16$) when larger simulations were feasible, and 128 hosts ($k = 8$) hosts for finer analyses.
- VL2 topology [19]. We used 1000 hosts with 50 ToR, 20 aggregation, and 5 intermediate switches. Inter-switch links have 10 times the bandwidth of host-to-ToR links.
- Oversubscribed topologies, created by adding more hosts to edge/ToR switches in the above topologies. We used a 512-host, 4:1 oversubscribed FatTree network ($k = 8$).

All our networks were as large or larger than those used by Raiciu et al. [32] for their packet-level simulations. Unless otherwise specified, we used 1000-byte packets, 1Gbps links (10Gbps inter-switch links for VL2), queues of 100 packets, and 100µs-delay pipes between queues.

TCP NewReno variants. We noticed in our simulation experiments that flows between nearby hosts of a topology sometimes suffered abnormally low throughput, even though they did not noticeably affect the average. We traced this problem to the NewReno variant used by *htsim*, called Slow-but-Steady [15], which causes flows to remain in fast recovery for a very long time when network round-trip times are low, as in datacenters and especially between nearby hosts. RFC 2582 [15, §4] suggests an alternative variant of NewReno for such scenarios called Impatient. After switching to this variant, the low-throughput outliers disappeared.

7.2 LocalFlow achieves optimal throughput

7.2.1 MapReduce-style workloads

We ran LocalFlow on a 1024-host FatTree network using a random permutation traffic matrix of long flows, i.e., each host sends a flow to one other host chosen at random without replacement. This workload can be satisfied at maximum speed on a FatTree given its full bisection bandwidth. We used a scheduling interval of 50ms and a dup-ACK threshold of 10 to accommodate reordering; these parameters are discussed later. We also ran PacketScatter, ECMP, Hedera with a 50ms scheduling interval, and MPTCP with 4 and 8 subflows per flow. Note that 50ms is an extremely generous interval for Hedera's centralized scheduler, being one to two orders of magnitude smaller than what it can actually handle [3, 32].

Figure 4 shows the throughput of individual flows in increasing order, with the legend sorted by decreasing average throughput. As expected, LocalFlow achieves near-optimal throughput for all flows, matching the performance of PacketScatter to within 1.4%. LocalFlow's main benefit over PacketScatter is that it splits fewer flows when there are multiple flows per destination, as we show later. Although LocalFlow-NS attempts to distribute flows locally, it does not split flows and so cannot avoid downstream collisions. It is also particularly unlucky in this case, performing worse than ECMP (typically their performance is similar).

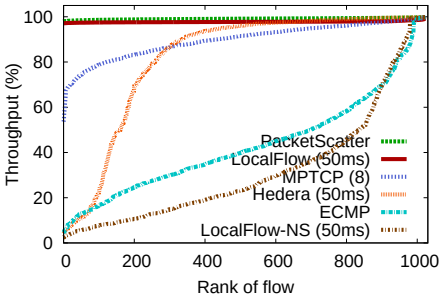


Figure 4: Individual flow throughputs for a random permutation on a 1024-host FatTree network.

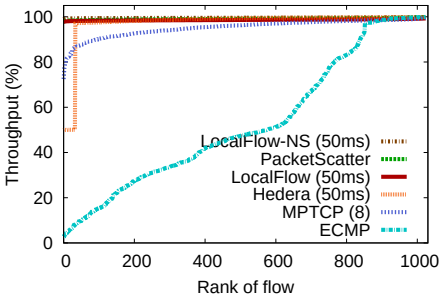


Figure 5: Individual flow throughputs for a stride permutation on a 1024-host FatTree network.

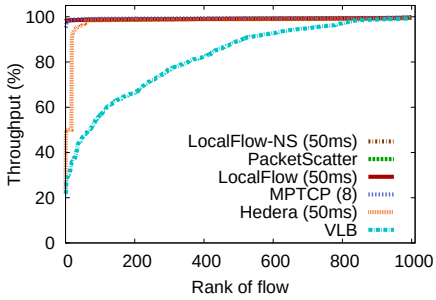


Figure 6: Individual flow throughputs for a random permutation on a 1000-host VL2 network.

MPTCP with 8 subflows achieves an average throughput that is 8.3% less than that of LocalFlow, and its slowest flow has 45% less throughput than that of LocalFlow. MPTCP with 4 subflows (not shown) performs substantially worse, achieving an average throughput that is 21% lower than LocalFlow. This is because there are fewer total subflows in the network; effectively, it throws fewer balls into the same number of bins. ECMP has the same problem but much worse because it throws N balls into N bins; this induces collisions with high probability, resulting in an average throughput that is 44% less than the optimal. For the remainder of our analysis, we use 8 subflows for MPTCP, the recommended number for datacenter settings [32].

Hedera's average throughput lies between MPTCP with 4 subflows and 8 subflows, but exhibits much higher variance. Although not shown, Hedera's variance was $\pm 28\%$, compared to $\pm 14\%$ for MPTCP (4). In general, Hedera does not cope well with a random permutation workload, which sends flows along different path lengths (most reach the core, some only reach aggregation, and a few only reach edge switches).

If instead we guarantee that all flows travel to the core before descending to their destinations, Hedera performs much better. Figure 5 shows the results of a stride$(N/2)$ permutation workload, where host i sends a flow to host $i+N/2$. All algorithms achieve higher throughput, and Hedera comes close to LocalFlow's performance, though its slowest flow has 49% less throughput than that of LocalFlow. Further, forcing all traffic to traverse the core incurs higher latency for potentially local, and yields performance in more oversubscribed settings. In fact, significant rack- or cluster-local communication is common in datacenter settings [5, 26], suggesting larger benefits for LocalFlow.

It may seem surprising that LocalFlow-NS has the highest average throughput in Figure 5, but this is due to the uniformity of the workload. LocalFlow-NS distributes the flows from each pod evenly over the core switches; since these flows target the same destination pod, the distribution is perfect. A similar effect arises when running a random permutation workload on the 1000-host VL2 topology, per Figure 6. In a VL2 network, aggregation and intermediate switches form a complete bipartite graph, thus it is only necessary to distribute the *number* of flows evenly over intermediate switches, which LocalFlow-NS does. In fact, LocalFlow-NS achieves optimal throughput for any permutation workload.

7.2.2 *Dynamic, heterogeneous workloads*

Real datacenters are typically oversubscribed, with hosts sending variable-sized flows to multiple destinations simultaneously. Using a 512-host, 4:1 oversubscribed FatTree network, we tested a realistic workload by having each host select a number of simultaneous flows to send from the VL2 dataset [19, Fig. 2],[3] with flow sizes also selected from this dataset. The flows ran in a closed loop, i.e., they restarted after finishing (with a new flow size). We ran LocalFlow with a 10ms scheduling interval and also allowed approximate splitting ($\delta > 0$). We used a 10ms scheduling interval for Hedera as well, even though this interval is one to two orders of magnitude smaller than Hedera can handle [3, 32]. Figure 7 shows results for the total throughput (total number of bytes transferred) and average flow completion times (which we discuss later in §7.7).

Total throughput, average flow completion time			
ECMP	0.0%, 0.0%	LF-1	+6.7%, −0.2%
Hedera	−7.2%, −17.0%	LF-1 ($\delta=0.01$)	+10.9%, −2.2%
MPTCP	+6.0%, +28.7%	LF-1 ($\delta=0.05$)	+7.2%, −1.0%
PS	+12.7%, +10.4%	LF-*NS* ($\delta=1$)	+6.6%, −1.9%

Figure 7: Total throughput and average flow completion time relative to ECMP, for a heterogeneous VL2 workload on a 512-host, 4:1 oversubscribed FatTree.

Using the VL2 distributions, there are over 12,000 simultaneous flows in the network. With this many flows, even ECMP's load-agnostic hashing should perform well due to averaging, and we expect all algorithms to deliver similar throughput; Figure 7 confirms this. Nevertheless, there are some interesting points to note.

First, LocalFlow-NS outperforms ECMP because it intelligently distributes flows, albeit locally. In fact, its performance is almost as good as LocalFlow due to the large number of flows. LocalFlow does not appear to gain much from exact splitting. We believe this is because over 86% of flows in the VL2 distribution are smaller than 125KB; such flows are small enough to complete within a 10ms interval, so it may be counterproductive to move or split them midstream. On the other hand, splitting too approximately ($\delta = 0.05$) also hurt LocalFlow's performance, because of the slight load imbalances incurred. $\delta = 0.01$ strikes the right balance for this workload, achieving close to PacketScatter's performance. All LocalFlow variants outperform MPTCP.

Hedera achieves 7.17% less throughput than ECMP. This is likely due to the small flows mentioned above, which are large enough to be scheduled by Hedera, but better left untouched. In addition, since Hedera reserves bandwidth along a flow's path, this bandwidth may go to waste after the flow completes but before the next scheduling interval starts.

7.3 LocalFlow handles failures gracefully

Although PacketScatter's throughput is competitive with LocalFlow above, this is not the case when network failures occur. As dis-

[3] We obtained the VL2 distributions by extracting plot data from the paper's PDF file.

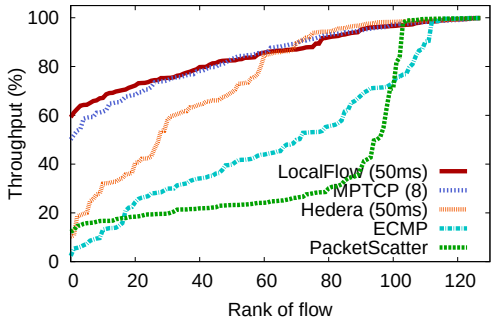


Figure 8: Individual flow throughputs for a random permutation on a 128-host FatTree network with failed links.

cussed in §4.3, if an entire switch fails, or if failures are spread uniformly throughout the network, PacketScatter is competitive with LocalFlow. However, if failures are skewed, as is expected in practice, PacketScatter's performance suffers drastically. Figure 8 shows the results of a random permutation on a 128-host FatTree network, when one aggregation switch (out of four) in each pod loses 3 of its 4 uplinks to the core. Upon learning of the failure, LocalFlow at the edge switches rebalances most outgoing traffic to the three other aggregation switches. From Figure 8, we see that LocalFlow and MPTCP deliver near-optimal throughput, whereas PacketScatter performs even worse than ECMP, achieving only 48% of the average throughput LocalFlow achieves.

7.4 LocalFlow uses little forwarding table space

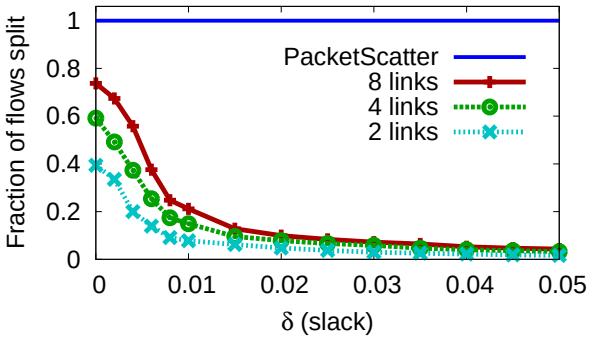
LocalFlow distributes the aggregate flow to each destination, so if several flows share the same destination, the number of subflows (splits) per flow is small. With approximate splitting, even fewer flows are split due to the added slack. This is important because splitting flows increases the size of a switch's forwarding tables.

To evaluate how much splitting LocalFlow does in practice, we ran our stand-alone simulator on a 3914-second TCP packet trace that saw 259,293 unique flows, collected from a 500-server university datacenter switch [5]. We used a scheduling interval of 50ms and different numbers of outgoing links, while varying δ. Figure 9 (top) shows these results as a function of δ. Although LocalFlow splits up to 78% of flows when $\delta = 0$ (using 8 links), this number drops to 21% when $\delta = 0.01$ and to 4.3% when $\delta = 0.05$. Thus, a slack of just 5% results in 95.7% of flows remaining unsplit! This is a big savings, because such flows do not require wildcard matching rules, and can thus be placed in an exact match table in the cheaper, more abundant, and more power-efficient SRAM of a switch.

The average number of subflows per flow similarly drops from 3.54 when $\delta = 0$ to 1.09 when $\delta = 0.05$ (note the minimum is 1 subflow per flow). This number more accurately predicts how much forwarding table space LocalFlow will use, since it counts the total number of rules required. Thus, using 8 links and $\delta = 0.05$, LocalFlow uses about 9% more forwarding table space than LocalFlow-NS, which only needs one rule per flow. Although PacketScatter creates almost 8 times as many subflows, recall that it only needs to store a small amount of state per destination, of which there are at most 500 in this dataset. As we will later see, PacketScatter pays for its excessive splitting in the form of longer flow completion times.

7.5 Smaller scheduling intervals improve performance, up to a limit

The previous experiments suggest that real workloads contain many short-lived flows. This is partly due to small flows, but even larger flows can complete in under a second in high-bandwidth data-



Avg. # of subflows/flow	2 links	4 links	8 links
LocalFlow ($\delta = 0$)	1.35	2.08	3.54
LocalFlow ($\delta = 0.01$)	1.06	1.20	1.49
LocalFlow ($\delta = 0.05$)	1.01	1.04	1.09
PacketScatter	2	4	8

Figure 9: Fraction of flows split (top) and average number of subflows per flow (bottom) by LocalFlow for different numbers of outgoing links, compared to other protocols, using a 3914-second trace from a real datacenter switch.

centers. In order to adapt quickly to these workloads, small scheduling intervals are necessary.

To measure the effect of scheduling interval size, we used a 128-host FatTree network running a random permutation with closed-loop flow arrivals. Flow sizes were selected from the VL2 dataset as before. Figure 10 shows the total throughput relative to ECMP for different scheduling intervals. Both Hedera and LocalFlow improve with smaller intervals, increasing 46% and 105%, respectively, as the interval is decreased from 1s to 1ms. LocalFlow's improvement is dramatic: it outperforms MPTCP at 10ms and, remarkably, outperforms PacketScatter at 1ms by over 7.7%. Hedera never outperforms MPTCP and its improvement is more gradual. This is partly due to the problem of overscheduling small flows, as we observed in Figure 7.[4] Of course, Hedera's centralized batch coordination makes such small intervals infeasible; Raiciu et al. experimentally evaluated Hedera with 5s intervals and argued analytically that, at best, 100ms intervals may be achievable.

The fact that LocalFlow outperforms PacketScatter is significant: it shows that splitting every flow can be *harmful*, since it exacerbates reordering. In contrast, LocalFlow may never even see a flow that starts and finishes within a scheduling interval. Since LocalFlow behaves more and more like PacketScatter as the scheduling interval tends to 0, it is important to place a lower bound on this interval, e.g., a few milliseconds.

7.6 Spatial splitting outperforms temporal

LocalFlow uses a precise, spatial splitting technique—based on either flowlets or counters—that is oblivious to the timing characteristics of a flow. Thus, it achieves precise load balancing despite traffic unpredictability, unlike temporal splitting techniques like FLARE. Load imbalances may arise in other techniques as well, such as Hedera's use of bandwidth reservations, or the random choices of stateless PacketScatter and its variants [13].

Our experiments showed that even slight load balances can significantly degrade throughput, especially for workloads that saturate the network's core. For example, we ran LocalFlow on a 128-host FatTree network using a random permutation, but chose an

[4] We note that our results are slightly different from those reported by Raiciu et al. [32, Fig. 13]. We believe this is due to their coarser approximation of the VL2 distribution.

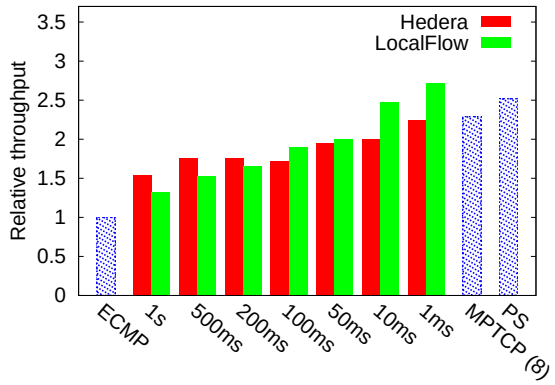


Figure 10: Total throughput on a random permutation with closed-loop flow arrivals, while varying the scheduling interval of LocalFlow and Hedera.

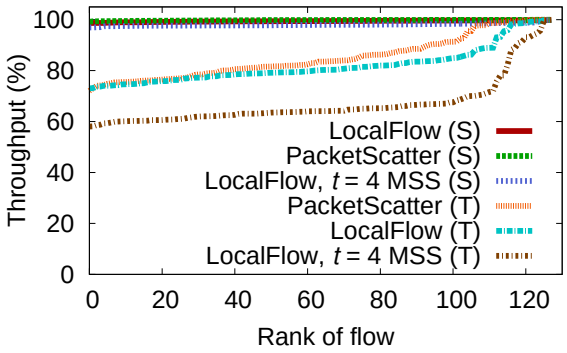


Figure 11: Throughputs for LocalFlow and PacketScatter flows using spatial (S) vs. temporal (T) splitting.

outgoing link at random for each packet (according to the splitting ratios), instead of based on the packet's sequence number. We also ran a stateless variant of PacketScatter, which selects a random outgoing link for each packet. Both of these schemes simulate temporal splitting because they achieve the desired splitting ratios on average in the long term, but due to randomness, exhibit load imbalances in the short term.

Figure 11 shows the results. The average throughput of flows using LocalFlow drops by 17% with temporal splitting; PacketScatter's performance drops by 14%. We also tested the effect of using a larger flowlet size with LocalFlow (the results that follow do not apply if counters are used to implement subflow splitting). Recall from §5.1 that flowlets facilitate finer splitting downstream, so higher switches in the FatTree should use smaller flowlets then lower switches. If instead we use a flowlet size of $t = 4$ MSS at all switches, LocalFlow's performance drops by 31% with temporal splitting. This is because the penalty of imprecise load balancing is higher when the scheduling unit is larger.

It is interesting that LocalFlow's performance itself drops slightly when using a larger flowlet. The reason for this is fundamental: even though splitting is spatial within a flow, the presence of *other* flows in the network introduces some temporal randomness. In fact, one of our modifications to *htsim* was to fix a bug in the existing implementation of PacketScatter [32], where an incorrect ordering of loops resulted in flowlets of size larger than 1 between edge/ToR and aggregation switches (instead of true packet spraying), causing similar performance degradations.

7.7 LocalFlow manages reordering and flow completion time better than PacketScatter

A major concern with splitting flows is that it may lead to increased packet reordering. Fortunately, we found that by simply increasing the duplicate-ACK threshold of end hosts' TCP, we could eliminate the adverse effects of reordering. Indeed, all of the experiments so far use a dup-ACK threshold of 10 for LocalFlow and PacketScatter, instead of the default of 3. One could also vary the threshold dynamically, as in RR-TCP [39], although we did not find this to be necessary in our experiments.

Although a higher dup-ACK threshold benefits both LocalFlow and PacketScatter, LocalFlow gains an advantage by splitting many fewer flows in practice. As Figure 9 shows, LocalFlow splits fewer than 4.3% of flows on a real datacenter switch trace. Put differently, over 95.7% of flows were not split and hence incurred no additional reordering. Further, small flows that complete inside a scheduling interval are not scheduled by LocalFlow; this gave LocalFlow a throughput advantage over PacketScatter in Figure 10, where the workload involved flow sizes from the VL2 distribution.

We now consider the approaches' flow completion time, which is sensitive to reordering. Recall that Figure 7 tests a heterogenous VL2 workload with thousands of simultaneous flows that are mostly smaller than 125KB. As the figure shows, the average flow completion time of all variants of LocalFlow is lower than ECMP, while delivering higher throughput. In contrast, although PacketScatter also delivers higher throughput, its average completion time is 10.4% *higher* than ECMP. MPTCP's performs even worse at 28.7% higher than ECMP, likely due to its overhead from splitting small flows. Hedera's numbers are interesting: it delivered lower throughput but achieved 17.0% lower completion time than ECMP. This is likely due to the fact that Hedera reserves bandwidth for a flow for the duration of a scheduling interval, ensuring that the flow completes quickly. However, since flows often complete before the scheduling interval ends, the reserved bandwidth may be wasted while waiting for the next interval to start.

7.8 Discussion: Comparing LocalFlow to end-host solutions

The previous experiments show that LocalFlow outperforms MPTCP. However, Raiciu et al. [32] describe three scenarios where MPTCP outperforms PacketScatter. Since PacketScatter can be viewed as a naïve version of LocalFlow, some of these criticisms may be relevant to LocalFlow, so we consider them here.

In the first scenario, one-third of the hosts in a 4:1 oversubscribed FatTree network send continuous flows, saturating the core, while the remaining hosts periodically send short flows *always using ECMP*. If a short flow intersects any of the paths used by PacketScatter for a flow, the entire flow backs off, whereas MPTCP's linked congestion control moves the flow to other paths. However, the assumption that short flows only use ECMP seems unjustified to us; PacketScatter (or LocalFlow) could split even these flows, resulting in superior performance to MPTCP. In fact, we saw this in Figures 7 and 10 using the VL2 workloads, which contain many small flows.

In the second scenario, a "hotspot" arises in the network due to a faulty link that operates at a lower rate (e.g., at 100Mbps instead of at 1Gbps). While slow or failed links may degrade PacketScatter's performance, LocalFlow can cope with these failures using the scheme described in 4.3.

The third scenario described are networks with variable-length paths. We agree that LocalFlow's performance may suffer in such settings. However, splitting flows over unequal paths has other problems not addressed in [32]; for example, it leads to increased queuing at the destination as packets from faster paths arrive out-

of-order. This consumes memory resources proportional to the flow rate times the latency difference between the paths.

MPTCP is suitable in more general settings than LocalFlow, as its co-designed congestion control adapts to asymmetric topologies, such as the wide-area Internet. However, for symmetric networks like in most datacenter environments, LocalFlow shows that a local algorithm can achieve near-optimal routing, and do so without modifying end hosts.

8. CONCLUSIONS

This paper introduces a practical, switch-local algorithm for routing traffic flows in datacenter networks in a load-aware manner. Compared to prior solutions, LocalFlow does not require centralized control, synchronization, or end-host modifications, while incurring modest forwarding table expansion. Perhaps more importantly, LocalFlow achieves optimal throughput in theory, and near-optimal throughput in practice, as our extensive simulation analysis shows. Our experiments revealed several interesting facts, such as the benefits of precise, spatial splitting over temporal splitting, and the impact of reordering on flow completion times.

Beyond its technical benefits, the design of LocalFlow illustrates two interesting research directions. First, rather than try to provide a general solution to the multi-commodity flow problem, LocalFlow takes advantage of certain symmetry properties of datacenter networks—namely, the balanced structure of fat-trees and other Clos topologies, both in terms of network distance and capacity. Second, LocalFlow can be realized on commodity hardware by leveraging emerging standards for configuring their data plane. While the most popular of these standards, OpenFlow, is designed for centralized network control, LocalFlow demonstrates using this programmability in a purely switch-local, or decentralized, fashion.

References

[1] OpenFlow Switch Specification, Version 1.3.0. `https://www.opennetworking.org/images/stories/downloads/specification/openflow-spec-v1.3.0.pdf`, 2012.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.

[4] B. Awerbuch and T. Leighton. A simple local-control approximation algorithm for multicommodity flow. In *FOCS*, 1993.

[5] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[6] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine grained traffic engineering for data centers. In *CoNext*, pages 8:1–8:12, 2011.

[7] J. E. Burns, T. J. Ott, A. E. Krzesinski, and K. E. Müller. Path selection and bandwidth allocation in MPLS networks. *Perform. Eval*, 52(2-3), 2003.

[8] M. Chiang, S. H. Low, A. Calderbank, and J. C. Doyle. Layering as optimization decomposition: A mathematical theory of network architectures. *Proceedings of the IEEE*, 95(1):255–312, 2007.

[9] Cisco Systems. Per-packet load balancing. `http://www.cisco.com/en/US/docs/ios/12_0s/feature/guide/pplb.pdf`, 2006.

[10] Cisco Systems. Application Extension Platform. `http://www.cisco.com/en/US/products/ps9701/`, 2011.

[11] C. Clos. A study of non-blocking switching networks. *Bell System Tech. Journal*, 32(2):406–424, 1953.

[12] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *SIGCOMM*, 2011.

[13] A. Dixit, P. Prakash, and R. R. Kompella. On the efficacy of fine-grained traffic splitting protocols in data center networks. Technical Report 11-011, Purdue University, 2011.

[14] S. Fischer, N. Kammenhuber, and A. Feldmann. REPLEX: Dynamic traffic engineering based on wardrop routing policies. In *CoNext*, 2006.

[15] S. Floyd and T. Henderson. The NewReno modification to TCP's fast recovery algorithm. RFC 2582, Network Working Group, 1999.

[16] N. Garg and J. Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM J. Comput.*, 37(2), 2007.

[17] N. Garg, V. V. Vazirani, and M. Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica*, 18(1), 1997.

[18] P. Geoffray and T. Hoefler. Adaptive routing strategies for modern high performance networks. In *HOT Interconnects*, 2008.

[19] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[20] J. He, M. Suchara, J. Rexford, and M. Chiang. From multiple decompositions to TRUMP: Traffic management using multipath protocol, 2008.

[21] C. Hopps. Analysis of an Equal-Cost Multi-Path algorithm. RFC 2992, Network Working Group, 2000.

[22] L. Jose, M. Yu, and J. Rexford. Online measurement of large traffic aggregates on commodity switches. In *HotICE*, 2011.

[23] Juniper. Junos SDK. `http://juniper.net/techpubs/en_US/release-independent/junos-sdk/`, 2011.

[24] S. Kandula, D. Katabi, B. S. Davie, and A. Charny. Walking the Tightrope: Responsive yet stable traffic engineering. In *SIGCOMM*, 2005.

[25] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Dynamic load balancing without packet reordering. *SIGCOMM Comp. Comm. Rev.*, 37, 2007.

[26] S. Kandula, S. Sengupta, A. G. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *IMC*, 2009.

[27] K.-C. Leung, V. O. K. Li, and D. Yang. An overview of packet reordering in Transmission Control Protocol (TCP): Problems, solutions, and challenges. *IEEE Trans. Parallel Distrib. Syst.*, 18, 2007.

[28] S. H. Low. A duality model of tcp and queue management algorithms. *Trans. Networking*, 11:525–536, 2002.

[29] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comp. Comm. Rev.*, 38, 2008.

[30] R. Ozdag. Intel ethernet switch FM6000 series – software defined networking, 2012.

[31] D. Palomar and M. Chiang. A tutorial on decomposition methods for network utility maximization. *IEEE JSAC*, 24(8):1439–1451, 2006.

[32] C. Raiciu, S. Barré, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath TCP. In *SIGCOMM*, 2011.

[33] S. Sen, S. Ihm, K. Ousterhout, and M. J. Freedman. Brief announcement: Bridging the theory-practice gap in multi-commodity flow routing. In *DISC*, 2011.

[34] J. Tourrilhes. OpenFlow 1.2 Proposal. `http://openflow.org/wk/index.php/OpenFlow_1_2_proposal`, 2011.

[35] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *STOC*, 1981.

[36] D. Wischik, C. Raiciu, A. Greenhalgh, , and M. Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *NSDI*, 2011.

[37] X. Wu and X. Yang. DARD: Distributed adaptive routing for datacenter networks. Technical Report TR-2011-01, Duke University, CS Dept. 2011.

[38] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. H. Katz. Detail: reducing the flow completion time tail in datacenter networks. In *SIGCOMM*, 2012.

[39] M. Zhang, B. Karp, S. Floyd, and L. L. Peterson. RR-TCP: A reordering-robust TCP with DSACK. In *ICNP*, 2003.