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. Even recent proposals like centralized scheduling or end-host multi-pathing give suboptimal throughput, and they suffer from poor scalability and other limitations.

We present a simple, switch-local algorithm called LocalFlow that is optimal (under standard assumptions), scalable, and practical. Although LocalFlow may split an individual flow (this is necessary for optimality), it does so infrequently by considering the *aggregate* flow per destination and allowing *slack* in distributing this flow. We use an optimization decomposition to prove LocalFlow's optimality when combined with unmodified end hosts' TCP. Splitting flows presents several new technical challenges that must be overcome in order to interact efficiently with TCP and work on emerging standards for programmable, commodity switches.

Since LocalFlow acts independently on each switch, it is highly scalable, adapts quickly to dynamic workloads, and admits flexible deployment strategies. We present detailed packet-level simulations comparing LocalFlow to a variety of alternative schemes, on real datacenter workloads.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Routing protocols*

Keywords

Flow routing; Datacenter networks; Local algorithms; Optimization decomposition

1. INTRODUCTION

The growth of popular Internet services and cloud-based platforms has spurred the construction of large-scale datacenters containing (hundreds of) thousands of servers, leading to a rash of research proposals for new datacenter networking architectures. Many such architectures (e.g., [1, 20]) are based on Clos topologies [13]; 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) [23] and Valiant Load Balancing (VLB) [42]. More recently, the networking community has proposed a series of load-aware solutions including both centralized solutions (e.g., [2, 7]), where routing decisions are made by a global scheduler, and distributed solutions, where routing decisions are made by end hosts (e.g., [26, 43]) or switches (e.g., [27]).

As we discuss in §2, these approaches have limitations. Centralized solutions like Hedera [2] face serious scalability challenges with today's datacenter workloads [6, 28]. End-host solutions like MPTCP [43] offer greater parallelism but cannot predict downstream collisions, forcing them to continuously react to congestion. Switch-local solutions like FLARE [27] scale well but are ill-suited to datacenters.

Most of the existing solutions do not split flows across multiple paths, making them necessarily (and significantly) suboptimal [18], as our evaluation confirms. But splitting flows is problematic in practice because it causes packet reordering, which in the case of TCP may lead to throughput collapse [30]. Solutions that do split flows are either load agnostic [11, 15], suboptimal and complicated [19, 43], or rely on specific traffic patterns [27].

We argue that switch-local solutions hold the best promise for handling today's high-scale and dynamic datacenter traffic patterns optimally. We present 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 [1, 20, 31]) and real deployments satisfy the symmetry property. Our optimality proof decomposes the MCF problem into two components, one of which is essentially solved by end hosts' TCP, while the other component is solved locally at each switch by LocalFlow. In fact, a naïve scheme called PacketScatter [11, 15], which essentially round-robins packets over a switch's outgoing links, also solves the latter component. However, PacketScatter is load agnostic: it splits *every* flow, which causes packet reordering and increases flow completion times, and it does not handle network failures well.

Current affiliations: *Microsoft Research, †Google, Inc.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s). Copyright is held by the author/owner(s).
CoNEXT'13, December 9-12, 2013, Santa Barbara, California, USA.
ACM 978-1-4503-2101-3/13/12.
http://dx.doi.org/10.1145/2535372.2535397.

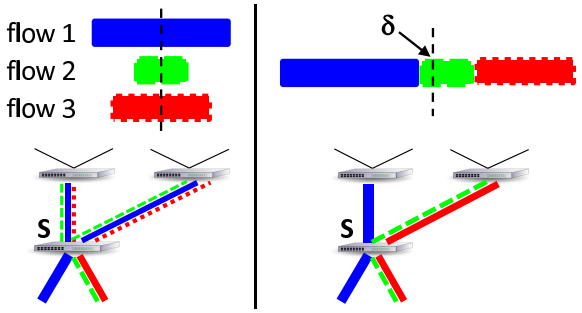


Figure 1: A set of flows to the same destination arrives at switch S. PacketScatter (left) splits every flow, whereas LocalFlow (right) distributes the aggregate flow, and only splits an individual flow if the load imbalance exceeds δ.

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 (typically <12). 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. 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 deep. Because it requires no 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, from running locally on each switch's CPU, to running on a single server for the entire network. In all cases, the scheduling performed for each switch is *independent* of the others.

Splitting flows introduces several technical challenges in order to achieve high accuracy, use modest forwarding table state, and interact properly with TCP. (Although we focus on TCP, we also discuss how to use LocalFlow with UDP traffic.) Besides splitting infrequently, LocalFlow employs two novel techniques to split flows efficiently. First, it splits individual 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 [34, 37]. Given the forthcoming nature of one of these features, and our desire to evaluate LocalFlow at large scale, our evaluation focuses on simulations. We use a full packet-level network simulator [43] as well as real datacenter traces [6, 20].

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, it split less than 4.3% of flows on a real switch packet trace and achieved 11% lower flow completion times. By modestly increasing the duplicate-ACK threshold of end hosts' TCP, LocalFlow avoids the adverse effects of packet reordering. Interestingly, the high accuracy of its spatial splitting is crucial, as even slight load imbalances 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 [20].

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 and evaluate it in §7. We address some deployment concerns in §8 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 rates achieved relative to optimal routing.

Centralized solutions typically run a sequential algorithm at a single server [2, 7, 9]. 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 serious problem [7, 20]. The rise of switches with externally-managed forwarding tables, such as OpenFlow [34, 37], has enabled solutions that operate at faster timescales. For example, Hedera's scheduler runs every 5 seconds, with the potential to run at subsecond intervals [2], and MicroTE's scheduler runs each second [7]. But recent studies [6, 28] 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 [14]. This infeasibility motivated our pursuit of a parallel solution.

End-host solutions employ more parallelism, and most give provable guarantees. TeXCP [26] and TRUMP [22] dynamically load-balance traffic over multiple paths between pairs of ingress-egress routers (e.g., MPLS tunnels) established by an underlying routing architecture. DARD [44] is a similar solution for datacenter networks that controls paths from end hosts. (We discuss MPTCP further below.) These solutions explicitly model paths in their formulation, though they limit the number of paths per source-destination pair to avoid exponential representation and convergence issues. 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 do not need to model or rely on per-path statistics. For example, REPLEX [16] gathers (aggregate) path information using measurements on adjacent links and by exchanging messages between neighbors.

None of the above solutions split individual flows, however, and hence cannot produce optimal results, since the unsplittable MCF problem is NP-hard and admits no constant-factor approximation [18]. MPTCP [39, 43] is an end-host solution that splits a flow into subflows and balances load across the subflows via linked congestion control. It uses two levels of sequence numbers and buffering to handle reordering across subflows. DeTail [45] modifies switches to do per-packet adaptive load balancing based on queue occupancy, in order to reduce the tail of flow completion times. It relies on layer-2 backpressure and modifications to end hosts' TCP to avoid congestion and handle reordering. Geoffray and Hoefler [19] propose an adaptive source-routing scheme that uses layer-2 back-

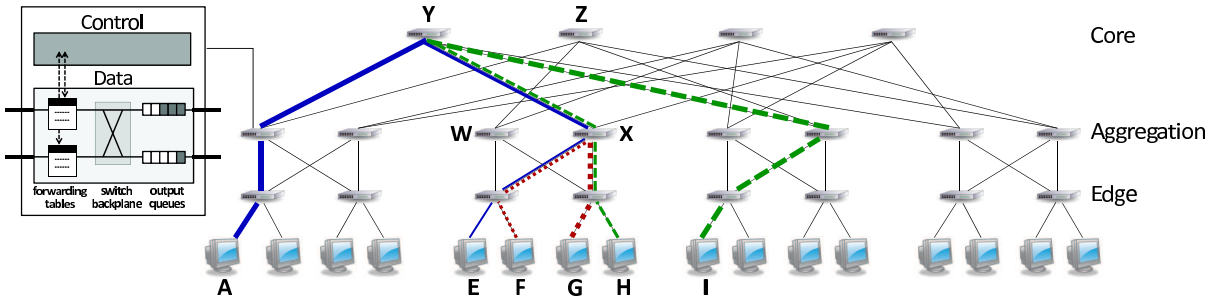


Figure 2: A FatTree network with 4-port switches. VL2 is a variation on this topology. End hosts A, G, I simultaneously transmit to E, F, H and collide at switches Y and X, but there is sufficient capacity to route all flows at full rate.

pressure and probes to evaluate alternative paths. Like DeTail, their scheme requires modifications to both end hosts and switches. FLARE [27] is a technique for splitting flows in wide-area networks that can be combined with systems like TeXCP. It exploits delays between packet bursts to route each burst along a different path while avoiding reordering.

It is instructive to compare our solution, LocalFlow, to the above schemes. Like all of them, LocalFlow splits individual flows, but whereas the above schemes tend to split *every* flow, LocalFlow splits very few flows. 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 unlike FLARE it splits flows spatially (e.g., based on TCP sequence numbers) and does not make timing assumptions. Our simulations show that spatial splitting significantly outperforms temporal splitting. Finally, LocalFlow achieves near-optimal routing in practice, which the other solutions have not demonstrated, despite being considerably simpler than them.

The only flow-splitting scheme that is simpler than LocalFlow is PacketScatter [11, 15], which we discussed earlier. LocalFlow can be viewed as a load-aware, efficient version of PacketScatter. We compare the schemes in detail while deriving LocalFlow's design.

There is a long history of theoretical algorithms for the MCF problem in each of the settings above: from centralized (e.g., [33]), to end-host (e.g., [4]), to switch-local (e.g., [3]). Although these algorithms give provably exact or approximate solutions, they are disconnected from practice for reasons we have previously outlined [40]. For example, they assume flows can be split arbitrarily (and instantaneously) at any node in the network, whereas this is difficult to do in practice. LocalFlow, in contrast, achieves 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 of this network (§3.1), outline the main scheduling loop for each switch (§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 [34, 37], Juniper NOS [25], or Cisco AXP [12].

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, resources, and programmability of switches are continually increasing [35, 38]. 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 scheduler [6, 28]. 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 scheduler with OpenFlow switches or a separate blade in the switch chassis of Juniper NOS or Cisco AXP switches. Alternatively, the number of these schedulers can be reduced and their locations changed to suit the network's scalability requirements. In the limit, a single centralized scheduler may be used. Regardless of the deployment strategy, LocalFlow's independence allows each switch to be scheduled by separate threads, processes, cores, or devices.

3.2 Main LocalFlow scheduling loop

LocalFlow runs a continuous scheduling loop for each switch. At the beginning of every interval, it:

1) Measures 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) Runs the scheduling algorithm using the flow rates from step 1 as input.
3) Updates 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. Note that Step 1 relies on measurements from the previous interval to inform scheduling decisions in the current interval. Although traffic patterns may change between intervals, LocalFlow's design copes well with this unpredictability, as we shall see.

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), all switches 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 these 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., [1, 20, 31]), which are Clos topologies [13] arranged as multi-rooted trees. FatTree [1] is a three-stage fat-tree network built using identical k-port switches arranged into three levels—edge, aggregation, and core—that supports full bisection bandwidth between $k^3/4$ end hosts. Figure 2 shows a 16-host FatTree network ($k = 4$). F10 [31] is a recent variant of FatTree that skews the connections between switch levels to achieve better fault tolerance, but is still symmetric by our definition. VL2 [20] 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. All of these networks can be oversubscribed by connecting more hosts to each edge/ToR switch, which preserves 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 scheduling loop (§3.2). At a high-level, LocalFlow attempts to find the optimal flow routing for the following maximum MCF problem:

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

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

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

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

This formulation reflects the complementary roles LocalFlow and TCP play. LocalFlow balances the flow rates $f^{s,d}_{u,v}$ across links (u,v) between adjacent switches, for a fixed set of commodity send rates x_i. This technique is similar to, but more aggressive than, the original link-balancing technique of Awerbuch and Leighton [5]. The intuition 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. In a symmetric network with a fixed set of send rates, this is equivalent to minimizing the maximum link utilization: $\min \max_{(u,v)\in E} \frac{\sum_{s,d} f^{s,d}_{u,v}}{c_{u,v}}$.

Link balancing on its own does not guarantee an optimal solution to the maximum MCF objective (1), which depends on the per-commodity (concave) utility functions U_i. Fortunately, LocalFlow can rely on the end hosts' TCP congestion control for this purpose. Using an idealized fluid model, it can be shown [32] that assuming backlogged senders (i.e., senders have more data to send) and given a fixed routing matrix, TCP, in its various forms, maximizes the total network utility. 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.

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 network failures and asymmetry (§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; it has been supported by Cisco switches for over a decade now [11]. Recent work by Dixit et al. [15] 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).

Although PacketScatter routes flows optimally, because it unconditionally splits *every* flow at individual-packet boundaries, 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 sending the same flow to 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 and 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 (1).

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). When the function returns, the total flow to the destination has been evenly distributed.

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). (For simplicity, we overload δ to mean either this fraction or the actual flow rate it corresponds to, depending on context.) 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 (and hence scheduled) by the algorithm, and only the last such flow entering a bin may give rise to an imbalance. After BINPACK returns, LOCALFLOW ensures 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 proved 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, whereas PacketScatter splits every flow.

Since LocalFlow may split a flow over a subset of the outgoing links, possibly unevenly, we cannot use the (load-agnostic) round-

```
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.

robin scheme of PacketScatter to implement Split. 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. These rules, along with their current rates (as measured in Step 1 of the main scheduling loop), comprise the input set F to function LocalFlow. Multi-resolution splitting is discussed in §5.

Even though LocalFlow's splitting strategy is load aware, it still uses local measurements to balance load proactively, which allows it to cope with traffic burstiness and unpredictability.

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 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 2 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. Also, since PacketScatter splits every flow, more flows are likely to be affected by a single link failure. 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 schedulers can learn about it from the underlying link-state protocol (which automatically disseminates this connectivity information). The upstream schedulers 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 this rebalancing may not be optimal. In general, determining the optimal rebalancing strategy requires non-local knowledge because the network is now asymetric. A scheme similar to the above can be used to run LocalFlow in an asymmetric network.

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 [37]. Figure 3 illustrates each type of rule. These rules represent single flows and subflows (§5.1), but they can also represent "metaflows" (§5.2), i.e., groups of flows to the same destination.

Since metaflow and subflow rules use partial wildcard matching, they must appear in the TCAM of a switch, which is scarcer and less power-efficient than SRAM. However, our simulations show that LocalFlow splits very few flows in practice, so only a few TCAM rules are needed; single-flow rules can be placed in SRAM.

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 single flow into two or more subflows, we use one of two techniques. Although these techniques may not be supported by current OpenFlow switches, the latest specifications [37] suggest that the functionality will appear soon. 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. 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 flowlets is spatial and thus different from that of FLARE [27], 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. For example, 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.[1] 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. 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.

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 com-

[1] Since TCP sequence numbers represent a byte offset, the bit string should actually start after bit $\lceil \lg(t \times \text{MSS}) \rceil$, where MSS is the maximum size of a TCP segment.

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).

mon in OpenFlow switches [37]. 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 inside a flow. 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, saving forwarding table space, as 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 distributes all flows to target E over three links with ratios $(1/4, 1/4, 1/2)$. The "all" rule is placed on the bottom to illustrate its lower priority (it captures the remaining $1/2$), although in OpenFlow priorities are explicit.

If there is a diversity of flows and source ports, this scheme splits the total flow rate by the desired ratios (approximately). But it may not do so if the distribution of source ports or flow sizes is unfavorable: for example, if there is a single large flow and many small flows. In such situations, metaflow rules can be combined with subflow rules for better accuracy. For example, a metaflow rule can be split using the subflow splitting technique. Note that this simultaneously splits all flows that match the rule.

We did not use metaflow rules in our evaluation since we found LocalFlow's space utilization to be modest in practice (§7.4).

6. ANALYSIS

We begin by analyzing the local (per-switch) complexity of LocalFlow, and then prove its optimality.

During each round, LocalFlow 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 $\delta > 0$, $O(|F|\log|F| + |F||L|\log|L|)$ steps are executed, 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, LocalFlow maintains at least one rule per flow; this can be reduced to one rule per destination by using metaflows. Both of these numbers increase when flow rules are split. 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) in an idealized fluid model [10]. In the remainder of this section, we refer to this idealized Network Utility Maximization (NUM) model when discussing the properties of TCP with respect to the MCF problem. Since LocalFlow and TCP alternately optimize their respective variables, we first show that the "master" LocalFlow optimization adapts link flow rates $f^d_{u,v}$ to minimize the maximum link cost $\frac{\sum_d f^d_{u,v}}{c_{u,v}}$, for the commodity send rates x^*_i determined by the "slave" TCP sub-problem. Then we examine the optimality conditions for TCP and show how the "link-balanced" flow rates determined by LocalFlow lead to an optimal solution to our original MCF objective.

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

PROOF. 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 would intersect with a shortest path and thus add to its 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 minimum-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 flow rate than 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. □

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

PROOF. To show that LocalFlow's "link-balanced" flow rates enable TCP to maximize the maximum MCF objective (1), we turn to the node-centric NUM formulation [10] of the TCP sub-problem, adapted for the multi-path setting to allow flow splitting.

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

Here, LocalFlow has already computed the set of flow variables $f^d_{u,v}$, which have been absorbed into the path probabilities π^p_i. Each

π_i^p 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 d. Since these variables are derived from the link flow rates, they implicitly satisfy the original MCF flow and send rate constraints (1).

To examine the effect of LocalFlow on the MCF objective, we focus on the optimality conditions for the TCP sub-problem, which is solved using dual decomposition [10]. In this approach, we first form the Lagrangian $L(x,\lambda)$ by introducing dual variables $\lambda_{u,v}$, one for each link constraint.

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

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 [32], and f is differentiable and invertible. Most TCP utilities (e.g., log) fall in this category [32]. Next, we construct the Lagrange dual function $Q(\lambda)$ maximized with respect to x_i:

$$x_i^* = f^{-1}(\beta) \text{ when } \frac{\partial L}{\partial x_i} = 0\ \forall x_i, \beta = \sum_p \pi_i^p \sum_{(u,v)\in p} \lambda_{u,v} \quad (2)$$

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

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

$$\sum_i f^{-1}(\beta) \sum_{p:(u,v)\in p} \pi_i^p = c_{u,v} \text{ when } \frac{\partial Q}{\lambda_{u,v}} = 0, \quad \forall (u,v) \in E \quad (3)$$

When (3) is satisfied, the system has reached the maximum network utility (1). TCP computes this solution in a distributed fashion using gradient ascent. 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:

$$\sum_i f^{-1}(\beta) \sum_{p:(u,v)\in p} \pi_i^p = \sum_i f^{-1}(\beta) \sum_{p:(u,w)\in p} \pi_i^p \quad (4)$$

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 (3) by ensuring that the per-commodity values of (4) are equal $\forall i$. PacketScatter satisfies this trivially by splitting every commodity evenly across the equal-cost links ($f_{u,v}^{s,d} = f_{u,w}^{s,d}$), resulting in equal link probabilities.

LocalFlow, on the other hand, groups commodities by destination when balancing the 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 reachable via the links are the same as well. Thus, if we group the commodities in (3) by destination d then the condition is satisfied when:

$$\sum_{i:s\to d} f^{-1}(\beta) \sum_{p:(u,v)\in p} \pi_i^p = \sum_{i:s'\to d} f^{-1}(\beta) \sum_{p:(u,w)\in p} \pi_i^p, \quad \forall t$$

Since LocalFlow distributes the per-destination flow (x_i^*) evenly across equal-cost links, i.e., $f_{u,v}^d = f_{u,w}^d \forall d$, we have:

$$\sum_{i:s\to d} x_i^* \sum_{p:(u,v)\in p} \pi_i^p = \sum_{i:s'\to d} x_i^* \sum_{p:(u,w)\in p} \pi_i^p \quad (5)$$

By substituting in (2), we arrive at the per-destination optimality condition for (3). Note that LocalFlow will continue to adjust flow rates to achieve (5) in response to TCP's optimized send rates (and vice-versa). On each iteration, LocalFlow minimizes the maximum link utilization by balancing per-destination link flow rates, opening up additional head room on each link for the current commodity send-rates x_i^* to grow. Between LocalFlow iterations, the TCP sub-problem maximizes its send rate objective to consume the additional capacity. Given a proper timescale separation between the "master" and "slave" problems, the distributed convex optimization process converges [10] to an optimal network utility. □

In practice, a proper timescale separation is on the order of a few RTTs, which in total is still <1ms for a typical datacenter. Thus, LocalFlow can safely use a scheduling interval of 10ms or greater. In general, the speed of convergence to the optimum depends on the variant of TCP being used.

Note that the individual send rate utility functions U_i can differ by source. This corresponds to end-hosts using different TCP variants (e.g., Cubic, NewReno) or even their own application-level congestion control over UDP. As long as the utility functions are concave and the send rates are elastic and not unbounded or static, i.e., they can adjust to link congestion back pressure, then the optimal MCF conditions under LocalFlow hold. If "fairness" between send rates is an issue, then the provider must ensure that the end-hosts employ some form of fairness-inducing congestion control (e.g., $U_i = \log \forall i$) [29].

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)

We use different techniques to evaluate LocalFlow's performance, including analysis (§6) 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 scenarios at a scale larger than any testbed we could construct, and to facilitate comparison with prior work. In fact, we used the same simulator codebase as MPTCP [39, 43], allowing direct comparisons.

7.1 Experimental setup

Simulations. We developed two simulators for LocalFlow. The first is a stand-alone simulator that runs LocalFlow on `pcap` packet traces. We used the packet traces collected by Benson et al. [6] from a university datacenter switch. To stress the algorithm, we simulated the effect of larger flows by constraining the link bandwidth of the switch.

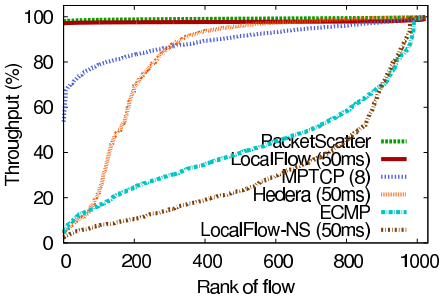


Figure 4: Individual flow throughputs for a random permutation on a 1024-host FatTree network.

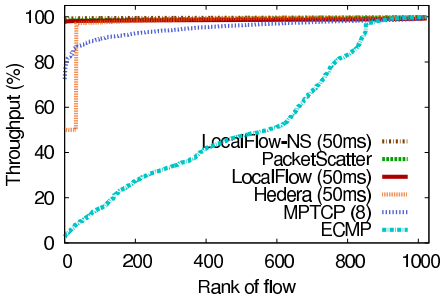


Figure 5: Individual flow throughputs for a stride permutation on a 1024-host FatTree network.

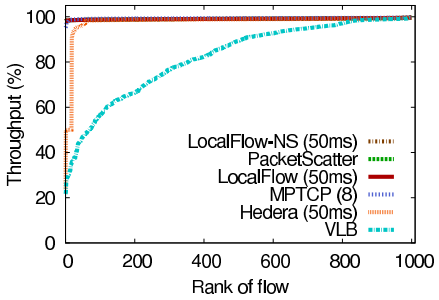


Figure 6: Individual flow throughputs for a random permutation on a 1000-host VL2 network.

Our second simulator is based on *htsim*, a full packet-level network simulator written by Raiciu et al. [39, 43]. 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 [2]. We modified and extended *htsim* to implement the LocalFlow algorithm. Notably, we added a switch abstraction that groups queues together and maintains a forwarding table based on the multi-resolution splitting rules defined in §5.

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 (e.g., by writing to `/proc/sys/` on Linux).

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

- FatTree topology built from k-port switches [1]. We used 1024 hosts ($k = 16$) when larger simulations were feasible, and 128 hosts ($k = 8$) hosts for finer analyses.
- VL2 topology [20]. 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. [39] 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 delays 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 [17], 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 [17] 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. Given its topology, a FatTree network can run this workload at full bisection bandwidth. We used a scheduling interval of 50ms and increased the dup-ACK threshold 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 optimistic interval for Hedera's centralized scheduler, being one to two orders of magnitude smaller than what it can actually handle [2, 39].

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).

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 [39].

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 with 4 subflows. 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 communication, and yields worse performance in more oversubscribed settings. In fact, signif-

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

icant rack- or cluster-local communication is common in datacenter settings [6, 28], 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 [20, Fig. 2][2], 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, which again is extremely optimistic. 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).

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 hurts 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, as Raiciu et al. [39] observed, Hedera optimistically reserves bandwidth along a flow's path assuming the flow can fill it, but this bandwidth may go to waste in the current scheduling interval if the flow is unable to do so.

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

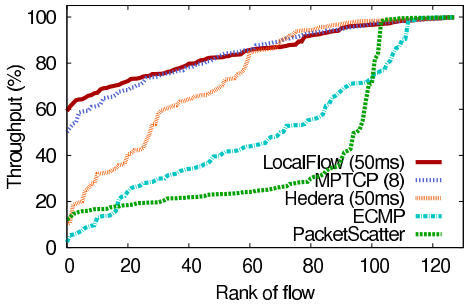


Figure 8: Individual flow throughputs for a random permutation on a 128-host FatTree network with failed links.

7.3 LocalFlow handles failures gracefully

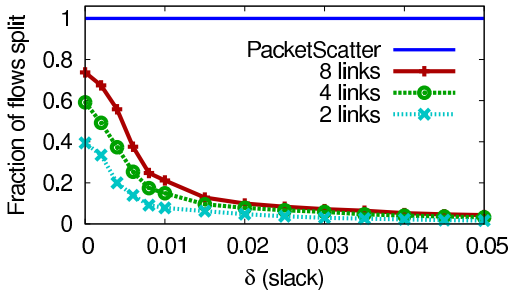
Although PacketScatter's throughput is competitive with LocalFlow above, this is not the case when network failures occur. As discussed in §4.3, if an entire switch fails, PacketScatter is competitive with LocalFlow, but if failures are skewed (as one would in practice), PacketScatter's performance suffers 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 of LocalFlow.

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 [6]. 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 more abundant and 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, it only needs to store a small amount of state per destination, of which there are at most 500 in this dataset. Later, we will see that PacketScatter pays for its excessive splitting in the form of longer flow completion times.



Avg. # of subflows/flow	2 links	4 links	8 links
LocalFlow (δ=0)	1.35	2.08	3.54
LocalFlow (δ=0.01)	1.06	1.20	1.49
LocalFlow (δ=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.

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 datacenters. 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.[3] 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 does not split flows that start and finish within a scheduling interval.

7.6 Spatial splitting outperforms temporal

LocalFlow uses a precise, spatial technique to split an individual flow—based on either flowlets or counters—that is oblivious to the timing characteristics of the flow. Thus, it achieves accurate 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 [15].

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

[3] We note that our results are slightly different from those reported by Raiciu et al. [39, Fig. 13]. We believe this is due to their coarser approximation of the VL2 distribution.

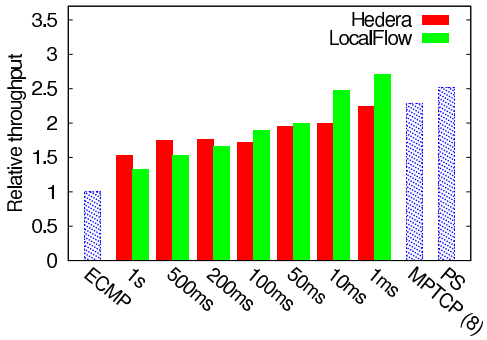


Figure 10: Total throughput on a random permutation with closed-loop flow arrivals, while varying the scheduling interval of LocalFlow and Hedera.

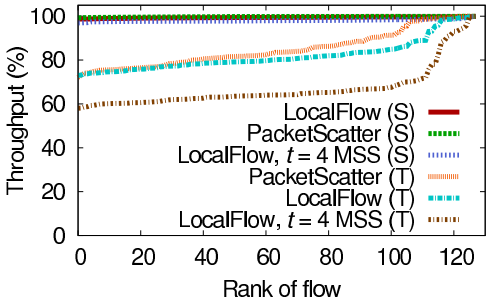


Figure 11: Throughputs for LocalFlow and PacketScatter flows using spatial (S) vs. temporal (T) splitting.

128-host FatTree network using a random permutation, but chose an 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 than 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. This causes occasional bursts of flowlets from different flows on the same outgoing link. In fact, one of our modifications to *htsim* was to fix a bug in the existing implementation of PacketScatter, 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 handles reordering and 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 dup-ACK threshold of end hosts' TCP, we could eliminate the adverse effects of reordering. For example, in the previous experiments, using a dup-ACK threshold of >15 for LocalFlow and PacketScatter (instead of the default of 3) is sufficient. One could also vary the threshold dynamically, as in RR-TCP [46], 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 split 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 flow completion time, which is sensitive to reordering. Recall that Figure 7 tests a heterogeneous 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 performs even worse at 28.7% higher than ECMP, likely due to its overhead from splitting small flows.

8. DEPLOYMENT CONCERNS

We discuss some considerations that should be made before deploying LocalFlow in a real datacenter network. These involve the network architecture and hardware, end-host TCP, and traffic workload. Some of these issues have been discussed in previous sections; we include them here for completeness.

Network architecture. LocalFlow is designed for symmetric networks. If the target network is asymmetric, LocalFlow may not achieve optimal routing. Asymmetry can arise by design, for example if the network has variable-length paths (e.g., BCube [21]), or it may arise due to hardware failures, such as a faulty link that operates at a lower rate (e.g., 100Mbps instead of 1Gbps). Raiciu et al. [39] analyzed these scenarios and showed that MPTCP's linked congestion control adapts well to asymmetry. LocalFlow can also cope with failures and asymmetry (see §4.3), but cannot guarantee good performance in these settings. Splitting a flow over unequal paths poses additional concerns not addressed in [39]; for example, it requires more buffering at the destination as packets from faster paths arrive out-of-order. This consumes memory proportional to the flow rate times the latency difference between the paths.

Switch implementation. The choice of switch implementation is critical to LocalFlow's performance. While implementing LocalFlow in a software switch is relatively straightforward,[4] limitations in software packet processing produce suboptimal results. Unlike hardware line-rate forwarding, software switching requires the NIC to interrupt the processor when packets are available. Under heavy load, the operating system uses interrupt coalescing to reduce signal handling overhead. However, this induces unpredictable forwarding delay which can increase packet reordering for split flows. Flow-steering in multi-queue NICs exacerbates the problem by directing flows to different CPU cores, leading to further packet interleaving. This is why our design assumes hardware switches.

Some recent studies [14, 24] paint a bleak picture of the performance of hardware OpenFlow switches, complaining that they handle only tens of flow setups per second and have scarce, power-hungry TCAMs. However, both these switches and the OpenFlow specification become faster and support more complex features every year. For example, high-end NoviFlow switches [35] have over 1 million TCAM entries, handle 1000 flow setups per second, and support OpenFlow 1.3 [37]. Though more expensive than commodity switches, they indicate that hardware support for LocalFlow's splitting techniques is around the corner. Recall that LocalFlow only requires TCAM entries for the few flows that are actually split.

End-host TCP. Since LocalFlow may cause additional packet reordering in flows that are split, the dup-ACK threshold of end-host TCP must be increased (as in our simulations) to avoid spurious fast retransmissions and congestion window collapse. Since TCP counts the *number* of reordered packets, the faster the network, the higher this threshold needs to be. Through its decomposition of the MCF problem, LocalFlow is compatible with other variants of TCP, such as those that use DSACK to detect spurious congestion signals and/or adjust the dup-ACK threshold dynamically [8, 46].

Workloads. As we have explained, LocalFlow's design enables it to cope with highly dynamic traffic patterns, even though it's scheduling decisions are based on measurements from the previous interval. By setting the δ slack appropriately, small, latency-sensitive flows can be efficiently scheduled without ever being split.

Since LocalFlow relies on TCP to adjust flow send rates, the presence of unregulated traffic such as UDP flows can degrade its optimality guarantee. UDP does not perform congestion control or guarantee ordered delivery. LocalFlow can accommodate the latter problem by preventing UDP flows from being split—for example, by using a policy in function BINPACK that places (unsplit) UDP flows before (splittable) TCP flows. However, the former problem is fundamental and means that a UDP flow can easily gain more than its fair share of bandwidth. This affects any flow routing scheme, including MPTCP, Hedera, and ECMP. To co-exist fairly with TCP flows, UDP flows must either be subject to user-space congestion control, encapsulated by an aggregate end-host TCP flow (as in Seawall [41]), or placed in per-flow or per-class switch queues that are rate-limited along its path. The latter is a straightforward extension of LocalFlow, but such quality-of-service controls have limited support in commodity switches.

9. 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.

10. ACKNOWLEDGMENTS

We thank Kay Ousterhout for her work on an initial simulator for the LocalFlow algorithm. We thank Jennifer Rexford, our shepherd, and the anonymous reviewers for their valuable feedback. This work was supported by funding from the National Science Foundation and a Google PhD Fellowship.

[4] Modifying the Open vSwitch [36] software switch to split flows using the TCP sequence number technique requires less than 100 LoC.

References

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

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

[3] B. Awerbuch and R. Khandekar. Distributed network monitoring and multicommodity flows: A primal-dual approach. In *PODC*, 2007.

[4] B. Awerbuch and R. Khandekar. Greedy distributed optimization of multi-commodity flows. *Distrib. Comput.*, 21(5), 2009.

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

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

[7] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine grained traffic engineering for data centers. In *CoNEXT*, 2011.

[8] S. Bhandarkar, A. L. N. Reddy, M. Allman, and E. Blanton. Improving the robustness of TCP to non-congestion events. RFC 4653, Internet Engineering Task Force, 2006.

[9] 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.

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

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

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

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

[14] 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.

[15] A. Dixit, P. Prakash, Y. Hu, and R. Kompella. On the impact of packet spraying in data center networks. In *INFOCOM*, 2013.

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

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

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

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

[20] 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.

[21] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, 2009.

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

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

[24] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. In *HotSDN*, 2013.

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

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

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

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

[29] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan. Rate control for communication networks: Shadow prices, proportional fairness and stability. *J. Oper. Res. Soc.*, 49:237–252, 1998.

[30] 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.

[31] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. In *NSDI*, 2013.

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

[33] A. Madry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *STOC*, 2010.

[34] 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.

[35] NoviSwitch. `http://noviflow.com/products/noviswitch/`, 2013.

[36] Open vSwitch. `http://openvswitch.org/`, 2012.

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

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

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

[40] 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.

[41] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: Performance isolation for cloud datacenter networks. In *HotCloud*, 2010.

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

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

[44] X. Wu and X. Yang. DARD: Distributed adaptive routing for datacenter networks. In *ICDCS*, 2012.

[45] 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.

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