Hedera: Dynamic Flow Scheduling for Data Center Networks

Mohammad Al-Fares[*]   Sivasankar Radhakrishnan[*]

Barath Raghavan[†]   Nelson Huang[*]   Amin Vahdat[*]

[*]{*malfares, sivasankar, nhuang, vahdat*}@cs.ucsd.edu   [†]*barath@cs.williams.edu*

[*]*Department of Computer Science and Engineering*   [†]*Department of Computer Science*
University of California, San Diego   *Williams College*

Abstract

Today's data centers offer tremendous aggregate bandwidth to clusters of tens of thousands of machines. However, because of limited port densities in even the highest-end switches, data center topologies typically consist of multi-rooted trees with many equal-cost paths between any given pair of hosts. Existing IP multipathing protocols usually rely on per-flow static hashing and can cause substantial bandwidth losses due to long-term collisions.

In this paper, we present Hedera, a scalable, dynamic flow scheduling system that adaptively schedules a multi-stage switching fabric to efficiently utilize aggregate network resources. We describe our implementation using commodity switches and unmodified hosts, and show that for a simulated 8,192 host data center, Hedera delivers bisection bandwidth that is 96% of optimal and up to 113% better than static load-balancing methods.

1   Introduction

At a rate and scale unforeseen just a few years ago, large organizations are building enormous data centers that support tens of thousands of machines; others are moving their computation, storage, and operations to cloud-computing hosting providers. Many applications—from commodity application hosting to scientific computing to web search and MapReduce—require substantial *intra-cluster* bandwidth. As data centers and their applications continue to scale, scaling the capacity of the network fabric for potential all-to-all communication presents a particular challenge.

There are several properties of cloud-based applications that make the problem of data center network design difficult. First, data center workloads are *a priori* unknown to the network designer and will likely be variable over both time and space. As a result, static resource allocation is insufficient. Second, customers wish to run their software on commodity operating systems; therefore, the network must deliver high bandwidth without requiring software or protocol changes. Third, virtualization technology—commonly used by cloud-based hosting providers to efficiently multiplex customers across physical machines—makes it difficult for customers to have guarantees that virtualized instances of applications run on the same physical rack. Without this physical locality, applications face inter-rack network bottlenecks in traditional data center topologies [2].

Applications alone are not to blame. The routing and forwarding protocols used in data centers were designed for very specific deployment settings. Traditionally, in ordinary enterprise/intranet environments, communication patterns are relatively predictable with a modest number of popular communication targets. There are typically only a handful of paths between hosts and secondary paths are used primarily for fault tolerance. In contrast, recent data center designs *rely* on the path multiplicity to achieve horizontal scaling of hosts [3, 16, 17, 19, 18]. For these reasons, data center topologies are very different from typical enterprise networks.

Some data center applications often initiate connections between a diverse range of hosts and require significant aggregate bandwidth. Because of limited port densities in the highest-end commercial switches, data center topologies often take the form of a multi-rooted tree with higher-speed links but decreasing aggregate bandwidth moving up the hierarchy [2]. These multi-rooted trees have many paths between all pairs of hosts. A key challenge is to simultaneously and dynamically forward flows along these paths to minimize/reduce link oversubscription and to deliver acceptable aggregate bandwidth.

Unfortunately, existing network forwarding protocols are optimized to select a single path for each source/destination pair in the absence of failures. Such static single-path forwarding can significantly underutilize multi-rooted trees with any fanout. State of the art forwarding in enterprise and data center environments

uses ECMP [21] (Equal Cost Multipath) to statically stripe flows across available paths using flow hashing. This static mapping of flows to paths does not account for either current network utilization or flow size, with resulting collisions overwhelming switch buffers and degrading overall switch utilization.

This paper presents Hedera, a dynamic flow scheduling system for multi-stage switch topologies found in data centers. Hedera collects flow information from constituent switches, computes non-conflicting paths for flows, and instructs switches to re-route traffic accordingly. Our goal is to maximize aggregate network utilization—bisection bandwidth—and to do so with minimal scheduler overhead or impact on active flows. By taking a global view of routing and traffic demands, we enable the scheduling system to see bottlenecks that switch-local schedulers cannot.

We have completed a full implementation of Hedera on the PortLand testbed [29]. For both our implementation and large-scale simulations, our algorithms deliver performance that is within a few percent of optimal—a hypothetical non-blocking switch—for numerous interesting and realistic communication patterns, and deliver in our testbed up to 4X more bandwidth than state of the art ECMP techniques. Hedera delivers these bandwidth improvements with modest control and computation overhead.

One requirement for our placement algorithms is an accurate view of the demand of individual flows under ideal conditions. Unfortunately, due to constraints at the end host or elsewhere in the network, measuring current TCP flow bandwidth may have no relation to the bandwidth the flow could achieve with appropriate scheduling. Thus, we present an efficient algorithm to estimate idealized bandwidth share that each flow would achieve under max-min fair resource allocation, and describe how this algorithm assists in the design of our scheduling techniques.

2 Background

The recent development of powerful distributed computing frameworks such as MapReduce [8], Hadoop [1] and Dryad [22] as well as web services such as search, e-commerce, and social networking have led to the construction of massive computing clusters composed of commodity-class PCs. Simultaneously, we have witnessed unprecedented growth in the size and complexity of datasets, up to several petabytes, stored on tens of thousands of machines [14].

These cluster applications can often be bottlenecked on the network, not by local resources [4, 7, 9, 14, 16]. Hence, improving application performance may hinge on improving network performance. Most traditional
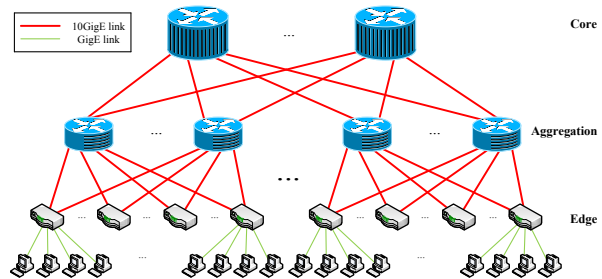

Figure 1: A common multi-rooted hierarchical tree.

data center network topologies are hierarchical trees with small, cheap edge switches connected to the end-hosts [2]. Such networks are interconnected by two or three layers of switches to overcome limitations in port densities available from commercial switches. With the push to build larger data centers encompassing tens of thousands of machines, recent research advocates the horizontal—rather than vertical—expansion of data center networks [3, 16, 17]; instead of using expensive core routers with higher speeds and port-densities, networks will leverage a larger number of parallel paths between any given source and destination edge switches, so-called *multi-rooted tree* topologies (e.g. Figure 1).

Thus we find ourselves at an impasse—with network designs using multi-rooted topologies that have the potential to deliver full bisection bandwidth among all communicating hosts, but without an efficient protocol to forward data within the network or a scheduler to appropriately allocate flows to paths to take advantage of this high degree of parallelism. To resolve these problems we present the architecture of Hedera, a system that exploits path diversity in data center topologies to enable near-ideal bisection bandwidth for a range of traffic patterns.

2.1 Data Center Traffic Patterns

Currently, since no data center traffic traces are publicly available due to privacy and security concerns, we generate patterns along the lines of traffic distributions in published work to emulate typical data center workloads for evaluating our techniques. We also create synthetic communication patterns likely to stress data center networks. Recent data center traffic studies [4, 16, 24] show tremendous variation in the communication matrix over space and time; a typical server exhibits many small, transactional-type RPC flows (e.g. search results), as well as few large transfers (e.g. backups, backend operations such as MapReduce jobs). We believe that the network fabric should be robust to a range of communication patterns and that application developers should not be forced to match their communication patterns to
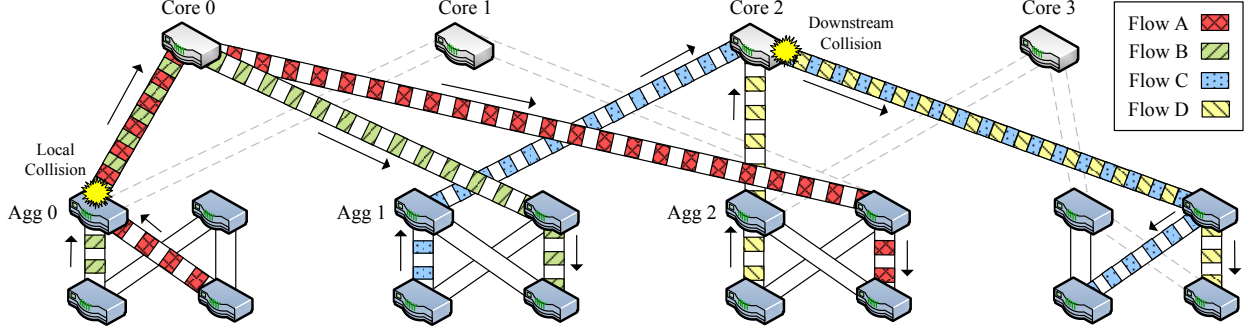
Figure 2: Examples of ECMP collisions resulting in reduced bisection bandwidth. Unused links omitted for clarity.

what may achieve good performance in a particular network setting, both to minimize development and debugging time and to enable easy porting from one network environment to another.

Therefore we focus in this paper on generating traffic patterns that stress and saturate the network, and comparing the performance of Hedera to current hash-based multipath forwarding schemes.

2.2 Current Data Center Multipathing

To take advantage of multiple paths in data center topologies, the current state of the art is to use Equal-Cost Multi-Path forwarding (ECMP) [2]. ECMP-enabled switches are configured with several possible forwarding paths for a given subnet. When a packet with multiple candidate paths arrives, it is forwarded on the one that corresponds to a hash of selected fields of that packet's headers modulo the number of paths [21], splitting load to each subnet across multiple paths. This way, a flow's packets all take the same path, and their arrival order is maintained (TCP's performance is significantly reduced when packet reordering occurs because it interprets that as a sign of packet loss due to network congestion).

A closely-related method is Valiant Load Balancing (VLB) [16, 17, 34], which essentially guarantees equal-spread load-balancing in a mesh network by bouncing individual packets from a source switch in the mesh off of randomly chosen intermediate "core" switches, which finally forward those packets to their destination switch. Recent realizations of VLB [16] perform randomized forwarding on a per-flow rather than on a per-packet basis to preserve packet ordering. Note that per-flow VLB becomes effectively equivalent to ECMP.

A key limitation of ECMP is that two or more large, long-lived flows can collide on their hash and end up on the same output port, creating an avoidable bottleneck as illustrated in Figure 2. Here, we consider a sample communication pattern among a subset of hosts in a multi-rooted, 1 Gbps network topology. We identify two types

of collisions caused by hashing. First, TCP flows A and B interfere locally at switch $Agg0$ due to a hash collision and are capped by the outgoing link's 1Gbps capacity to $Core0$. Second, with downstream interference, $Agg1$ and $Agg2$ forward packets independently and cannot foresee the collision at $Core2$ for flows C and D.

In this example, all four TCP flows could have reached capacities of 1Gbps with improved forwarding; flow A could have been forwarded to $Core1$, and flow D could have been forwarded to $Core3$. But due to these collisions, all four flows are bottlenecked at a rate of 500Mbps each, a 50% bisection bandwidth loss.
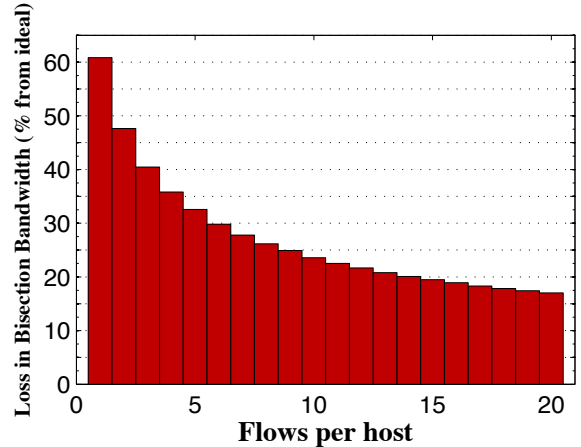

Figure 3: Example of ECMP bisection bandwidth losses vs. number of TCP flows per host for a k=48 fat-tree.

Note that the performance of ECMP and flow-based VLB intrinsically depends on flow size and the number of flows per host. Hash-based forwarding performs well in cases where hosts in the network perform all-to-all communication with one another simultaneously, or with individual flows that last only a few RTTs. Non-uniform communication patterns, especially those involving transfers of large blocks of data, require more careful scheduling of flows to avoid network bottlenecks.

We defer a full evaluation of these trade-offs to Section 6, however we can capture the intuition behind performance reduction of hashing with a simple Monte Carlo simulation. Consider a 3-stage fat-tree composed of 1GigE 48-port switches, with 27k hosts performing a data shuffle. Flows are hashed onto paths and each link is capped at 1GigE. If each host transfers an equal amount of data to all remote hosts one at a time, hash collisions will reduce the network's bisection bandwidth by an average of 60.8% (Figure 3). However, if each host communicates to remote hosts in parallel across 1,000 simultaneous flows, hash collisions will only reduce total bisection bandwidth by 2.5%. The intuition here is that if there are many simultaneous flows from each host, their individual rates will be small and collisions will not be significantly costly: each link has 1,000 slots to fill and performance will only degrade if substantially more than 1,000 flows hash to the same link. Overall, Hedera *complements* ECMP, supplementing default ECMP behavior for communication patterns that cause ECMP problems.

2.3 Dynamic Flow Demand Estimation

Figure 2 illustrates another important requirement for any dynamic network scheduling mechanism. The straightforward approach to find a good network-wide schedule is to measure the utilization of all links in the network and move flows from highly-utilized links to less utilized links. The key question becomes which flows to move. Again, the straightforward approach is to measure the bandwidth consumed by each flow on constrained links and move a flow to an alternate path with sufficient capacity for that flow. Unfortunately, a flow's current bandwidth may not reflect actual demand. We define a TCP flow's *natural* demand to mean the rate it would grow to in a fully non-blocking network, such that eventually it becomes limited by either the sender or receiver NIC speed. For example, in Figure 2, all flows communicate at 500Mbps, though all could communicate at 1Gbps with better forwarding. In Section 4.2, we show how to efficiently estimate the natural demands of flows to better inform Hedera's placement algorithms.

3 Architecture

Described at a high-level, Hedera has a control loop of three basic steps. First, it detects large flows at the edge switches. Next, it estimates the natural demand of large flows and uses placement algorithms to compute good paths for them. And finally, these paths are installed on the switches. We designed Hedera to support any general multi-rooted tree topology, such as the one in Figure 1, and in Section 5 we show our physical implementation using a fat-tree topology.

3.1 Switch Initialization

To take advantage of the path diversity in multi-rooted trees, we must spread outgoing traffic to or from any host as evenly as possible among all the core switches. Therefore, in our system, a packet's path is non-deterministic and chosen on its way up to the core, and is deterministic returning from the core switches to its destination edge switch. Specifically, for multi-rooted topologies, there is exactly one active minimum-cost path from any given core switch to any destination host.

To enforce this determinism on the downward path, we initialize core switches with the prefixes for the IP address ranges of destination pods. A *pod* is any sub-grouping down from the core switches (in our fat-tree testbed, it is a complete bipartite graph of aggregation and edge switches, see Figure 8). Similarly, we initialize aggregation switches with prefixes for downward ports of the edge switches in that pod. Finally, edge switches forward packets directly to their connected hosts.

When a new flow starts, the default switch behavior is to forward it based on a hash on the flow's 10-tuple along one of its equal-cost paths (similar to ECMP). This path is used until the flow grows past a threshold rate, at which point Hedera dynamically calculates an appropriate placement for it. Therefore, all flows are assumed to be small until they grow beyond a threshold, 100 Mbps in our implementation (10% of each host's 1GigE link). *Flows* are packet streams with the same 10-tuple of <src MAC, dst MAC, src IP, dst IP, EtherType, IP protocol, TCP src port, dst port, VLAN tag, input port>.

3.2 Scheduler Design

A central scheduler, possibly replicated for fail-over and scalability, manipulates the forwarding tables of the edge and aggregation switches dynamically, based on regular updates of current network-wide communication demands. The scheduler aims to assign flows to non-conflicting paths; more specifically, it tries to not place multiple flows on a link that cannot accommodate their combined natural bandwidth demands.

In this model, whenever a flow persists for some time and its bandwidth demand grows beyond a defined limit, we assign it a path using one of the scheduling algorithms described in Section 4. Depending on this chosen path, the scheduler inserts flow entries into the edge and aggregation switches of the source pod for that flow; these entries redirect the flow on its newly chosen path. The flow entries expire after a timeout once the flow terminates. Note that the state maintained by the scheduler is only soft-state and does not have to be synchronized with any replicas to handle failures. Scheduler state is not re-

$$
\begin{bmatrix}
 & (\tfrac{1}{3})_1 & (\tfrac{1}{3})_1 & (\tfrac{1}{3})_1 \\
(\tfrac{1}{3})_2 & & (\tfrac{1}{3})_1 & 0_0 \\
(\tfrac{1}{2})_1 & 0_0 & & (\tfrac{1}{2})_1 \\
0_0 & (\tfrac{1}{2})_2 & 0_0 &
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
 & [\tfrac{1}{3}]_1 & (\tfrac{1}{3})_1 & (\tfrac{1}{3})_1 \\
[\tfrac{1}{3}]_2 & & (\tfrac{1}{3})_1 & 0_0 \\
[\tfrac{1}{3}]_1 & 0_0 & & (\tfrac{1}{2})_1 \\
0_0 & [\tfrac{1}{3}]_2 & 0_0 &
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
 & [\tfrac{1}{3}]_1 & (\tfrac{1}{3})_1 & (\tfrac{1}{3})_1 \\
[\tfrac{1}{3}]_2 & & (\tfrac{1}{3})_1 & 0_0 \\
[\tfrac{1}{3}]_1 & 0_0 & & (\tfrac{2}{3})_1 \\
0_0 & [\tfrac{1}{3}]_2 & 0_0 &
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
 & [\tfrac{1}{3}]_1 & (\tfrac{1}{3})_1 & [\tfrac{1}{3}]_1 \\
[\tfrac{1}{3}]_2 & & (\tfrac{1}{3})_1 & 0_0 \\
[\tfrac{1}{3}]_1 & 0_0 & & [\tfrac{2}{3}]_1 \\
0_0 & [\tfrac{1}{3}]_2 & 0_0 &
\end{bmatrix}
$$

Figure 4: An example of estimating demands in a network of 4 hosts. Each matrix element denotes demand per flow as a fraction of the NIC bandwidth. Subscripts denote the number of flows from that source (rows) to destination (columns). Entries in parentheses are yet to converge. Grayed out entries in square brackets have converged.

quired for correctness (connectivity); rather it aids as a performance optimization.

Of course, the choice of the specific scheduling algorithm is open. In this paper, we compare two algorithms, Global First Fit and Simulated Annealing, to ECMP. Both algorithms search for flow-to-core mappings with the objective of increasing the aggregate bisection bandwidth for current communication patterns, supplementing default ECMP forwarding for large flows.

4 Estimation and Scheduling

Finding flow routes in a general network while not exceeding the capacity of any link is called the MULTI-COMMODITY FLOW problem, which is NP-complete for integer flows [11]. And while simultaneous flow routing is solvable in polynomial time for 3-stage Clos networks, no polynomial time algorithm is known for 5-stage Clos networks (i.e. 3-tier fat-trees) [20]. Since we do not aim to optimize Hedera for a specific topology, this paper presents practical heuristics that can be applied to a range of realistic data center topologies.

4.1 Host- vs. Network-Limited Flows

A flow can be classified into two categories: network-limited (e.g. data transfer from RAM) and host-limited (e.g. limited by host disk access, processing, etc.). A network-limited flow will use all bandwidth available to it along its assigned path. Such a flow is limited by congestion in the network, not at the host NIC. A host-limited flow can theoretically achieve a maximum throughput limited by the "slower" of the source and destination hosts. In the case of non-optimal scheduling, a network-limited flow might achieve a bandwidth less than the maximum possible bandwidth available from the underlying topology. In this paper, we focus on network-limited flows, since host-limited flows are a symptom of intra-machine bottlenecks, which are beyond the scope of this paper.

4.2 Demand Estimation

A TCP flow's current sending rate says little about its natural bandwidth demand in an ideal non-blocking net-

work (Section 2.3). Therefore, to make intelligent flow placement decisions, we need to know the flows' max-min fair bandwidth allocation as if they are limited only by the sender or receiver NIC. When network limited, a sender will try to distribute its available bandwidth fairly among all its outgoing flows. TCP's AIMD behavior combined with fair queueing in the network tries to achieve max-min fairness. Note that when there are multiple flows from a host A to another host B, each of the flows will have the same steady state demand. We now describe how to find TCP demands in a hypothetical equilibrium state.

The input to the demand estimator is the set F of source and destination pairs for all active large flows. The estimator maintains an $N \times N$ matrix M; N is the number of hosts. The element in the ith row, j^{th} column contains 3 values: (1) the number of flows from host i to host j, (2) the estimated demand of each of the flows from host i to host j, and (3) a "converged" flag that marks flows whose demands have converged.

The demand estimator performs repeated iterations of increasing the flow capacities from the sources and decreasing exceeded capacity at the receivers until the flow capacities converge; Figure 7 presents the pseudocode. Note that in each iteration of decreasing flow capacities at the receivers, one or more flows converge until eventually all flows converge to the natural demands. The estimation time complexity is $O(|F|)$.

Figure 4 illustrates the process of estimating flow demands with a simple example. Consider 4 hosts (H_0, H_1, H_2 and H_3) connected by a non-blocking topology. Suppose H_0 sends 1 flow each to H_1, H_2 and H_3; H_1 sends 2 flows to H_0 and 1 flow to H_2; H_2 sends 1 flow each to H_0 and H_3; and H_3 sends 2 flows to H_1. The figure shows the iterations of the demand estimator. The matrices indicate the flow demands during successive stages of the algorithm starting with an increase in flow capacity from the sender followed by a decrease in flow capacity at the receiver and so on. The last matrix indicates the final estimated natural demands of the flows.

For real communication patterns, the demand matrix for currently active flows is a sparse matrix since most hosts will be communicating with a small subset of remote hosts at a time. The demand estimator is also

```
GLOBAL-FIRST-FIT(f: flow)
1       if f.assigned then
2           return old path assignment for f
3       foreach p ∈ P_src→dst do
4           if p.used + f.rate < p.capacity then
5               p.used ← p.used + f.rate
6               return p
7       else
8           h = HASH(f)
9           return p = P_src→dst(h)
```

Figure 5: Pseudocode for Global First Fit. GLOBAL-FIRST-FIT is called for each flow in the system.

largely parallelizable, facilitating scalability. In fact, our implementation uses both parallelism and sparse matrix data structures to improve the performance and memory footprint of the algorithm.

4.3 Global First Fit

In a multi-rooted tree topology, there are several possible equal-cost paths between any pair of source and destination hosts. When a new large flow is detected, (e.g. 10% of the host's link capacity), the scheduler linearly searches all possible paths to find one whose link components can all accommodate that flow. If such a path is found, then that flow is "placed" on that path: First, a capacity reservation is made for that flow on the links corresponding to the path. Second, the scheduler creates forwarding entries in the corresponding edge and aggregation switches. To do so, the scheduler maintains the reserved capacity on every link in the network and uses that to determine which paths are available to carry new flows. Reservations are cleared when flows expire.

Note that this corresponds to a first fit algorithm; a flow is greedily assigned the *first* path that can accommodate it. When the network is lightly loaded, finding such a path among the many possible paths is likely to be easy; however, as the network load increases and links become saturated, this choice becomes more difficult. Global First Fit does not guarantee that all flows will be accommodated, but this algorithm performs relatively well in practice as shown in Section 6. We show the pseudocode for Global First Fit in Figure 5.

4.4 Simulated Annealing

Next we describe the Simulated Annealing scheduler, which performs a probabilistic search to efficiently compute paths for flows. The key insight of our approach is to assign a single core switch for each destination host rather than a core switch for each flow. This reduces

```
SIMULATED-ANNEALING(n : iteration count)
1       s ← INIT-STATE()
2       e ← E(s)
3       s_B ← s, e_B ← e
4       T_0 ← n
5       for T ← T_0 ... 0 do
6           s_N ← NEIGHBOR(s)
7           e_N ← E(s_N)
8           if e_N < e_B then
9               s_B ← s_N, e_B ← e_N
10          if P(e, e_N, T) > RAND() then
11              s ← s_N, e ← e_N
12      return s_B
```

Figure 6: Pseudocode for Simulated Annealing. s denotes the current state with energy $E(s) = e$. e_B denotes the best energy seen so far in state s_B. T denotes the temperature. e_N is the energy of a neighboring state s_N.

the search space significantly. Simulated Annealing forwards all flows destined to a particular host A through the designated core switch for host A.

The input to the algorithm is the set of all large flows to be placed, and their flow demands as estimated by the demand estimator. Simulated Annealing searches through a solution state space to find a near-optimal solution (Figure 6). A function E defines the energy in the current state. In each iteration, we move to a neighboring state with a certain acceptance probability P, depending on the energies in the current and neighboring states and the current temperature T. The temperature is decreased with each iteration of the Simulated Annealing algorithm and we stop iterating when the temperature is zero. Allowing the solution to move to a higher energy state allows us to avoid local minima.

1. State s: A set of mappings from destination hosts to core switches. Each host in a pod is assigned a particular core switch that it receives traffic from.

2. Energy function E: The total exceeded capacity over all the links in the current state. Every state assigns a unique path to every flow. We use that information to find the links for which the total capacity is exceeded and sum up exceeded demands over these links.

3. Temperature T: The remaining number of iterations before termination.

4. Acceptance probability P for transition from state s to neighbor state s_n, with energies E and E_n.

$$P(E_n, E, T) = \begin{cases} 1 & \text{if } E_n < E \\ e^{c(E-E_n)/T} & \text{if } E_n \geq E \end{cases}$$

where c is a parameter that can be varied. We empirically determined that $c = 0.5 \times T_0$ gives best results for a 16 host cluster and $c = 1000 \times T_0$ is best for larger data centers.

5. Neighbor generator function NEIGHBOR(): Swaps the assigned core switches for a pair of hosts in any of the pods in the current state s.

While simulated annealing is a known technique, our contribution lies in an optimization to significantly reduce the search space and the choice of appropriate energy and neighbor selection functions to ensure rapid convergence to a near optimal schedule. A straightforward approach is to assign a core for each flow individually and perform simulated annealing. However this results in a huge search space limiting the effectiveness of simulated annealing. The diameter of the search space (maximum number of neighbor hops between any two states) with this approach is equal to the number of flows in the system. Our technique of assigning core switches to destination hosts reduces the diameter of the search space to the minimum of the number of flows and the number of hosts in the data center. This heuristic reduces the search space significantly: in a 27k host data center with 27k large flows, the search space size is reduced by a factor of 10^{12000}. Simulated Annealing performs better when the size of the search space and its diameter are reduced [12]. With the straightforward approach, the runtime of the algorithm is proportional to the number of flows and the number of iterations while our technique's runtime depends only on the number of iterations.

We implemented both the baseline and optimized version of Simulated Annealing. Our simulations show that for randomized communication patterns in a 8,192 host data center with 16k flows, our techniques deliver a 20% improvement in bisection bandwidth and a 10X reduction in computation time compared to the baseline. These gains increase both with the size of the data center as well as the number of flows.

Initial state: Each pod has some fixed downlink capacity from the core switches which is useful only for traffic destined to that pod. So an important insight here is that we should distribute the core switches among the hosts in a single pod. For a fat-tree, the number of hosts in a pod is equal to the number of core switches, suggesting a one-to-one mapping. We restrict our solution search space to such assignments, i.e. we assign cores not to individual flows, but to destination hosts. Note that this choice of initial state is only used when the Simulated Annealing scheduler is run for the first time. We use an optimization to handle the dynamics of the system which reduces the importance of this initial state over time.

```
ESTIMATE-DEMANDS()
1     for all i, j
2         M_{i,j} ← 0
3     do
4         foreach h ∈ H do EST-SRC(h)
5         foreach h ∈ H do EST-DST(h)
6     while some M_{i,j}.demand changed
7     return M

EST-SRC(src: host)
1     d_F ← 0
2     n_U ← 0
3     foreach f ∈ ⟨src → dst⟩ do
4         if f.converged then
5             d_F ← d_F + f.demand
6         else
7             n_U ← n_U + 1
8     e_S ← (1.0 − d_F) / n_U
9     foreach f ∈ ⟨src → dst⟩ and not f.converged do
10        M_{f.src,f.dst}.demand ← e_S

EST-DST(dst: host)
1     d_T, d_S, n_R ← 0
2     foreach f ∈ ⟨src → dst⟩
3         f.rl ← true
4         d_T ← d_T + f.demand
5         n_R ← n_R + 1
6     if d_T ≤ 1.0 then
7         return
8     e_S ← 1.0 / n_R
9     do
10        n_R ← 0
11        foreach f ∈ ⟨src → dst⟩ and f.rl do
12            if f.demand < e_S then
13                d_S ← d_S + f.demand
14                f.rl ← false
15            else
16                n_R ← n_R + 1
17        e_S ← (1.0 − d_S) / n_R
18    while some f.rl was set to false
19    foreach f ∈ ⟨src → dst⟩ and f.rl do
20        M_{f.src,f.dst}.demand ← e_S
21        M_{f.src,f.dst}.converged ← true
```

Figure 7: Demand estimator for TCP flows. M is the demand matrix and H is the set of hosts. d_F denotes "converged" demand, n_U is the number of unconverged flows, e_S is the computed equal share rate, and $\langle src \rightarrow dst \rangle$ is the set of flows from src to some dst. In EST-DST d_T is the total demand, d_S is sender limited demand, $f.rl$ is a flag for a receiver limited flow and n_R is the number of receiver limited flows.

Neighbor generator: A well-crafted neighbor generator function intrinsically avoids deep local minima. Complying with the idea of restricting the solution search space to mappings with near-uniform mapping of hosts in a pod to core switches, our implementation employs three different neighbor generator functions: (1) swap the assigned core switches for any two randomly chosen hosts in a randomly chosen pod, (2) swap the assigned core switches for any two randomly chosen hosts in a

randomly chosen edge switch, (3) randomly choose an edge or aggregation switch with equal probability and swap the assigned core switches for a random pair of hosts that use the chosen edge or aggregation switch to reach their currently assigned core switches. Our neighbor generator function randomly chooses between the 3 described techniques with equal probability at runtime for each iteration. Using multiple neighbor generator functions helps us avoid deep local minima in the search spaces of individual neighbor generator functions.

Calculation of energy function: The energy function for a neighbor can be calculated incrementally based on the energy in the current state and the cores that were swapped in the neighbor. We need not recalculate exceeded capacities for all links. Swapping assigned cores for a pair of hosts only affects those flows destined to those two hosts. So we need to recalculate the difference in the energy function only for those specific links involved and update the value of the energy based on the energy in the current state. Thus, the time to calculate the energy only depends on the number of large flows destined to the two affected hosts.

Dynamically changing flows: With dynamically changing flow patterns, in every scheduling phase, a few flows would be newly classified as large flows and a few older ones would have completed their transfers. We have implemented an optimization where we set the initial state to the best state from the previous scheduling phase. This allows the route-placement of existing, continuing flows to be disrupted as little as possible if their current paths can still support their bandwidth requirements. Further, the initial state that is used when the Simulated Annealing scheduler first starts up becomes less relevant over time due to this optimization.

Search space: The key characteristic of Simulated Annealing is assigning unique core switches based on destination hosts in a pod, crucial to reducing the size of the search space. However, there are communication patterns where an optimal solution necessarily requires a single destination host to receive incoming traffic through multiple core switches. While we omit the details for brevity, we find that, at least for the fat tree topology, all communication patterns can be handled if: i) the maximum number of large flows to or from a host is at most $k/2$, where k is the number of ports in the network switches, or ii) the minimum threshold of each large flow is set to $2/k$ of the link capacity. Given that in practice data centers are likely to be built from relatively high-radix switches, e.g., $k \geq 32$, our search space optimization is unlikely to eliminate the potential for locating optimal flow assignments in practice.

Algorithm Complexity	Time	Space
Global First-Fit	$O((k/2)^2)$	$O(k^3 + |F|)$
Simulated Annealing	$O(f_{avg})$	$O(k^3 + |F|)$

Table 1: Time and Space Complexity of Global First Fit and Simulated Annealing. k is the number of switch ports, $|F|$ is the total number of large flows, and f_{avg} is the average number of large flows to a host. The k^3 factor is due to in-memory link-state structures, and the $|F|$ factor is due to the flows' state.

4.5 Comparison of Placement Algorithms

With Global First Fit, a large flow can be re-routed immediately upon detection and is essentially pinned to its reserved links. Whereas Simulated Annealing waits for the next scheduling tick, uses previously computed flow placements to optimize the current placement, and delivers even better network utilization on average due to its probabilistic search.

We chose the Global First Fit and Simulated Annealing algorithms for their simplicity; we take the view that more complex algorithms can hinder the scalability and efficiency of the scheduler while gaining only incremental bandwidth returns. We believe that they strike the right balance of computational complexity and delivered performance gains. Table 1 gives the time and space complexities of both algorithms. Note that the time complexity of Global First Fit is independent of $|F|$, the number of large flows in the network, and that the time complexity of Simulated Annealing is independent of k.

More to the point, the simplicity of our algorithms makes them both well-suited for implementation in hardware, such as in an FPGA, as they consist mainly of simple arithmetic. Such an implementation would substantially reduce the communication overhead of crossing the network stack of a standalone scheduler machine.

Overall, while Simulated Annealing is more conceptually involved, we show in Sec. 6 that it almost always outperforms Global First Fit, and delivers close to the optimal bisection bandwidth both for our testbed and in larger simulations. We believe the additional conceptual complexity of Simulated Annealing is justified by the bandwidth gains and tremendous investment in the network infrastructure of modern data centers.

4.6 Fault Tolerance

Any scheduler must account for switch and link failures in performing flow assignments. While we omit the details for brevity, our Hedera implementation augments the PortLand routing and fault tolerance protocols [29]. Hence, the Hedera scheduler is aware of failures using the standard PortLand mechanisms and can re-route flows mapped to failed components.

5 Implementation

To test our scheduling techniques on a real physical multi-rooted network, we built as an example the fat-tree network described abstractly in prior work [3]. In addition, to understand how our algorithms scale with network size, we implemented a simulator to model the behavior of large networks with many flows under the control of a scheduling algorithm.

5.1 Topology

For the rest of the paper, we adopt the following terminology: for a fat-tree network built from k-port switches, there are k *pods*, each consisting of two layers: lower pod switches (*edge* switches), and the upper pod switches (*aggregation* switches). Each edge switch manages $(k/2)$ hosts. The k pods are interconnected by $(k/2)^2$ *core* switches.

One of the main advantages of this topology is the high degree of available path diversity; between any given source and destination host pair, there are $(k/2)^2$ equal-cost paths, each corresponding to a core switch. Note, however, that these paths are not link-disjoint. To take advantage of this path diversity (to maximize the achievable bisection bandwidth), we must assign flows non-conflicting paths. A key requirement of our work is to perform such scheduling with no modifications to end-host network stacks or operating systems. Our testbed consists of 16 hosts interconnected using a fat-tree of twenty 4-port switches, as shown in Figure 8.

We deploy a parallel control plane connecting all switches to a 48-port non-blocking GigE switch. We emphasize that this control network is not required for the Hedera architecture, but is used in our testbed as a debugging and comparison tool. This network transports only traffic monitoring and management messages to and from the switches; however, these messages could also be transmitted using the data plane. Naturally, for larger networks of thousands of hosts, a control network could be organized as a traditional tree, since control traffic should be only a small fraction of the data traffic. In our deployment, the flow scheduler runs on a separate machine connected to the 48-port switch.

5.2 Hardware Description

The switches in the testbed are 1U dual-core 3.2 GHz Intel Xeon machines, with 3GB RAM, and NetFPGA 4-port GigE PCI card switches [26]. The 16 hosts are 1U quad-core 2.13 GHz Intel Xeon machines with 3GB of RAM. These hosts have two GigE ports, the first connected to the control network for testing and debugging, and the other to its NetFPGA edge switch. The control
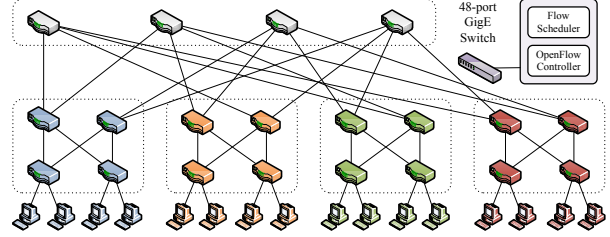

Figure 8: System Architecture. The interconnect shows the data-plane network, with GigE links throughout.

network is organized as a simple star topology. The central switch is a Quanta LB4G 48-port GigE switch. The scheduler machine has a dual-core 2.4 GHz Intel Pentium CPU and 2GB of RAM.

5.3 OpenFlow Control

The switches in the tree all run OpenFlow [27], which allows access to the forwarding tables for all switches. OpenFlow implementations have been ported to a variety of commercial switches, including those from Juniper, HP, and Cisco. OpenFlow switches match incoming packets to flow entries that specify a particular action such as duplication, forwarding on a specific port, dropping, and broadcast. The NetFPGA OpenFlow switches have 2 hardware tables: a 32-entry TCAM (that accepts variable-length prefixes) and a 32K entry SRAM that only accepts flow entries with fully qualified 10-tuples.

When OpenFlow switches start, they attempt to open a secure channel to a central controller. The controller can query, insert, modify flow entries, or perform a host of other actions. The switches maintain statistics per flow and per port, such as total byte counts, and flow durations. The default behavior of the switch is as follows: if an incoming packet does not match any of the flow entries in the TCAM or SRAM table, the switch inserts a new flow entry with the appropriate output port (based on ECMP) which allows any subsequent packets to be directly forwarded at line rate in hardware. Once a flow grows beyond the specified threshold, the Hedera scheduler may modify the flow entry for that flow to redirect it along a newly chosen path.

5.4 Scheduling Frequency

Our scheduler implementation polls the edge switches for flow statistics (to detect large flows), and performs demand estimation and scheduling once every five seconds. This period is due entirely to a register read-rate limitation of the OpenFlow NetFPGA implementation. However, our scalability measurements in Section 6 show that a modestly-provisioned machine can schedule

tens of thousands of flows in a few milliseconds, and that even at the 5 second polling rate, Hedera significantly outperforms the bisection bandwidth of current ECMP methods. In general, we believe that sub-second and potentially sub-100ms scheduling intervals should be possible using straightforward techniques.

5.5 Simulator

Since our physical testbed is restricted to 16 hosts, we also developed a simulator that coarsely models the behavior of a network of TCP flows. The simulator accounts for flow arrivals and departures to show the scalability of our system for larger networks with dynamic communication patterns. We examine our different scheduling algorithms using the flow simulator for networks with as many as 8,192 hosts. Existing packet-level simulators, such as *ns-2*, are not suitable for this purpose: e.g. a simulation with 8,192 hosts each sending at 1Gbps would have to process 2.5×10^{11} packets for a 60 second run. If a per-packet simulator were used to model the transmission of 1 million packets per second using TCP, it would take 71 hours to simulate just that one test case.

Our simulator models the data center topology as a network graph with directed edges. Each edge has a fixed capacity. The simulator accepts as input a communication pattern among hosts and uses it, along with a specification of average flow sizes and arrival rates, to generate simulated traffic. The simulator generates new flows with an exponentially distributed length, with start times based on a Poisson arrival process with a given mean. Destinations are based upon the suite in Section 6.

The simulation proceeds in discrete time ticks. At each tick, the simulation updates the rates of all flows in the network, generates new flows if needed. Periodically it also calls the scheduler to assign (new) routes to flows. When calling the Simulated Annealing and Global First Fit schedulers, the simulator first calls the demand estimator and passes along its results.

When updating flow rates, the simulator models TCP slow start and AIMD, but without performing per-packet computations. Each tick, the simulator shuffles the order of flows and computes the expected rate increase for each flow, constrained by available bandwidth on the flow's path. If a flow is in slow start, its rate is doubled. If it is in congestion avoidance, its rate is additively increased (using an additive increase factor of 15 MB/s to simulate a network with an RTT of 100 μs). If the flow's path is saturated, the flow's rate is halved and bandwidth is freed along the path. Each tick, we also compute the number of bytes sent by the flow and purge flows that have completed sending all their bytes.

Since our simulator does not model individual packets, it does not capture the variations in performance of different packet sizes. Another consequence of this decision is that our simulation cannot capture inter-flow dynamics or buffer behavior. As a result, it is likely that TCP Reno/New Reno would perform somewhat *worse* than predicted by our simulator. In addition, we model TCP flows as unidirectional although real TCP flows involve ACKs in the reverse direction; however, for 1500 byte Ethernet frames and delayed ACKs, the bandwidth consumed by ACKs is about 2%. We feel these trade-offs are necessary to study networks of the scale described in this paper.

We ran each simulation for the equivalent of 60 seconds and measured the average bisection bandwidth during the middle 40 seconds. Since the simulator does not capture inter-flow dynamics and traffic burstiness our results are optimistic (simulator bandwidth exceeds testbed measurements) for ECMP based flow placement because resulting hash collisions would sometimes cause an entire window of data to be lost, resulting in a coarse-grained timeout on the testbed (see Section 6). For the control network we observed that the performance in the simulator more closely matched the performance on the testbed. Similarly, for Global First Fit and Simulated Annealing, which try to optimize for minimum contention, we observed that the performance from the simulator and testbed matched very well. Across all the results, the simulator indicated better performance than the testbed when there is contention between flows.

6 Evaluation

This section describes our evaluation of Hedera using our testbed and simulator. The goal of these tests is to determine the aggregate achieved bisection bandwidth with various traffic patterns.

6.1 Benchmark Communication Suite

In the absence of commercial data center network traces, for both the testbed and the simulator evaluation, we first create a group of communication patterns similar to [3] according to the following styles:

(1) Stride(i): A host with index x sends to the host with index $(x + i)mod(num\_hosts)$.

(2) Staggered Prob *(EdgeP, PodP)*: A host sends to another host in the same edge switch with probability *EdgeP*, and to its same pod with probability *PodP*, and to the rest of the network with probability *1-EdgeP - PodP*.

(3) Random: A host sends to any other host in the network with uniform probability. We include bijective mappings and ones where hotspots are present.
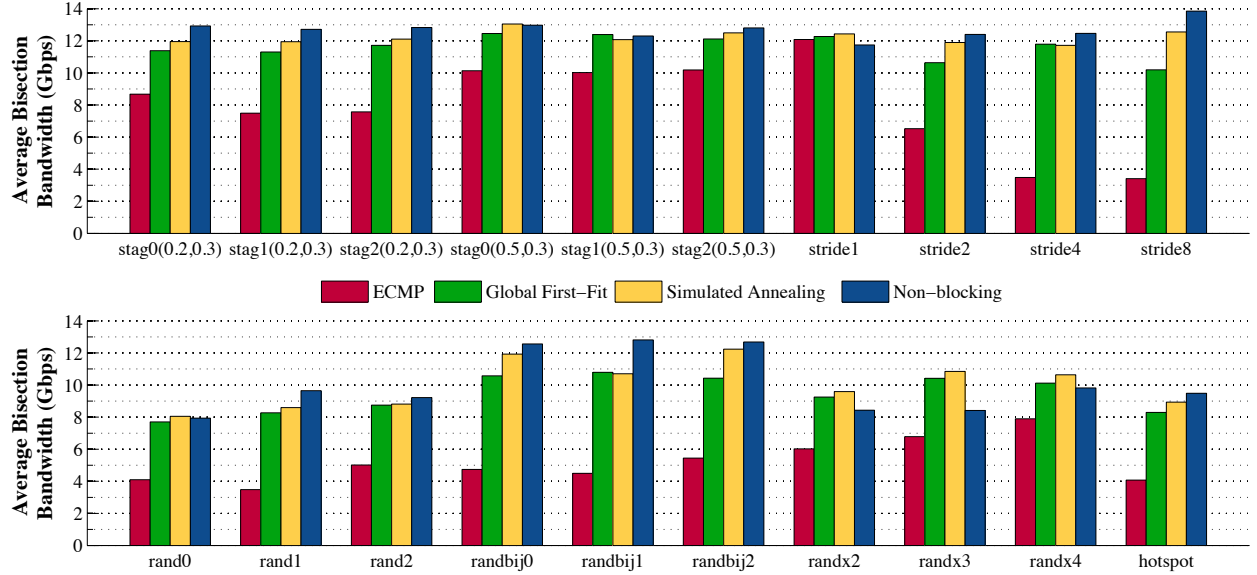
Figure 9: Physical testbed benchmark suite results for the three routing methods vs. a non-blocking switch. Figures indicate network bisection bandwidth achieved for staggered, stride, and randomized communication patterns.

We consider these mappings for networks of different sizes: 16 hosts, 1,024 hosts, and 8,192 hosts, corresponding to $k = \{4, 16, 32\}$.

6.2 Testbed Benchmark Results

We ran benchmark tests as follows: 16 hosts open socket sinks for incoming traffic and measure the incoming bandwidth constantly. The hosts in succession then start their flows according to the sizes and destinations as described above. Each experiment lasts for 60 seconds and uses TCP flows; we observed the average bisection bandwidth for the middle 40 seconds.

We compare the performance of the scheduler on the fat-tree network to that of the same experiments on the control network. The control network connects all 16 hosts using a non-blocking 48-port gigabit Ethernet switch and represents an ideal network. In addition, we include a static hash-based ECMP scheme, where the forwarding path is determined by a hash of the destination host IP address.

Figure 9 shows the bisection bandwidth for a variety of randomized, staggered, stride and hotspot communication patterns; our experiments saturate the links using TCP. In virtually all the communication patterns explored, Global First Fit and Simulated Annealing significantly outperform static hashing (ECMP), and achieve near the optimal bisection bandwidth of the network (15.4Gb/s goodput). Naturally, the performance of these schemes improves as the level of communication locality increases, as demonstrated by the staggered prob-

ability figures. Note that for stride patterns (common to HPC computation applications), the heuristics consistently compute the correct flow-to-core mappings to efficiently utilize the fat-tree network, whereas the performance of static hash quickly deteriorates as the stride length increases. Furthermore, for certain patterns, these heuristics also marginally outperform the commercial 48-port switch used for our control network. We suspect this is due to different buffers/algorithms of the Net-FPGAs vs. the Quanta switch.

Upon closer examination of the performance using packet captures from the testbed, we found that when there was contention between flows, an entire TCP window of packets was often lost. So the TCP connection was idle until the retransmission timer fired ($RTO_{min} = 200$ms). ECMP hash based flow placement experienced over 5 times the number of retransmission timeouts as the other schemes. This explains the overoptimistic performance of ECMP in the simulator as explained in Section 5 since our simulator does not model retransmission timeouts and individual packet losses.

6.3 Data Shuffle

We also performed an all-to-all in-memory data shuffle in our testbed. A data shuffle is an expensive but necessary operation for many MapReduce/Hadoop operations in which every host transfers a large amount of data to every other host participating in the shuffle. In this experiment, each host sequentially transfers 500MB to every other host using TCP (a 120GB shuffle).

	ECMP	GFF	SA	Control
Shuffle time (s)	438.44	335.50	335.96	306.37
Host completion (s)	358.14	258.70	261.96	226.56
Bisec. BW (Gbps)	2.811	3.891	3.843	4.443
Goodput (MB/s)	20.94	28.99	28.63	33.10

Table 2: A 120GB shuffle for the placement heuristics in our testbed. Shown is total shuffle time, average host-completion time, average bisection bandwidth and average host goodput.
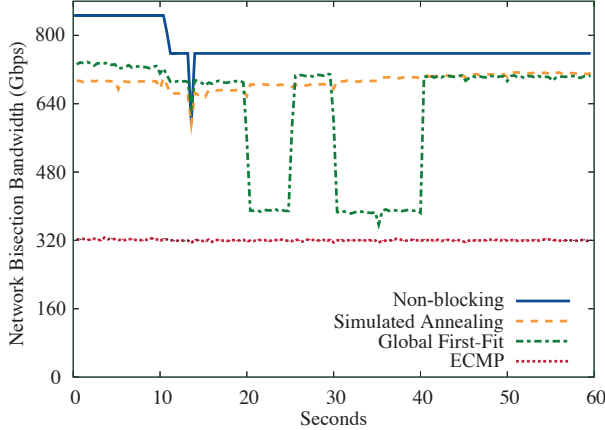

Figure 11: Network bisection bandwidth vs. time for a 1,024 host fat-tree and a random bijective traffic pattern.

The shuffle results in Table 2 show that centralized flow scheduling performs considerably better (39% better bisection bandwidth) than static ECMP hash-based routing. Comparing this to the data shuffle performed in VL2 [16], which involved all hosts making *simultaneous* transfers to all other hosts (versus the sequential transfers in our work), we see that static hashing performs better when the number of flows is significantly larger than the number of paths; intuitively a hash collision is less likely to introduce significant degradation when any imbalance is averaged over a large number of flows. For this reason, in addition to the delay of the Hedera observation/route-computation control loop, we believe that traffic workloads characterized by many small, short RPC-like flows would have limited benefit from dynamic scheduling, and Hedera's default ECMP forwarding performs load-balancing efficiently in this case. Hence, by thresholding our scheduler to only operate on larger flows, Hedera performs well for both types of communication patterns.

6.4 Simulation Results

6.4.1 Communication Patterns

In Figure 10 we show the aggregate bisection bandwidth achieved when running the benchmark suite for a simulated fat-tree network with 8,192 hosts (when k=32).

	Number of Hosts		
SA Iterations	16	1,024	8,192
1000	78.73	74.69	72.83
50000	78.93	75.79	74.27
100000	78.62	75.77	75.00
500000	79.35	75.87	74.94
1000000	79.04	75.78	75.03
1500000	78.71	75.82	75.13
2000000	78.17	75.87	75.05
Non-blocking	81.24	78.34	77.63

Table 3: Percentage of final bisection bandwidth by varying the Simulated Annealing iterations, for a case of random destinations, normalized to the full network bisection. Also shown is the same load running on a non-blocking topology.

We compare our algorithms against a hypothetical non-blocking switch for the entire data center and against static ECMP hashing. The performance of ECMP worsens as the probability of local communication decreases. This is because even for a completely fair and perfectly uniform hash function, collisions in path assignments *do* happen, either within the same switch or with flows at a downstream switch, wasting a portion of the available bandwidth. A global scheduler makes discrete flow placements that are chosen by design to reduce overlap. In most of these different communication patterns, our dynamic placement algorithms significantly outperform static ECMP hashing. Figure 11 shows the variation over time of the bisection bandwidth for the 1,024 host fat-tree network. Global First Fit and Simulated Annealing perform fairly close to optimal for most of the experiment.

6.4.2 Quality of Simulated Annealing

To explore the parameter space of Simulated Annealing, we show in Table 3 the effect of varying the number of iterations at each scheduling period for a randomized, non-bijective communication pattern. This table confirms our initial intuition regarding the assignment quality vs. the number of iterations, as most of the improvement takes place in the first few iterations. We observed that the performance of Simulated Annealing asymptotically approaches the best result found by Simulated Annealing after the first few iterations.

The table also shows the percentage of final bisection bandwidth for a random communication pattern as number of hosts and flows increases. This supports our belief that Simulated Annealing can be run with relatively few iterations in each scheduling period and still achieve comparable performance over time. This is aided by remembering core assignments across periods, and by the arrival of only a few new large flows each interval.
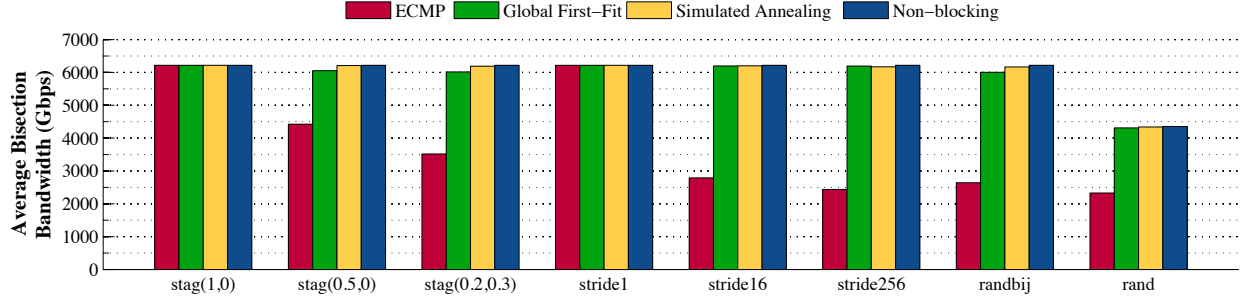
Figure 10: Comparison of scheduling algorithms for different traffic patterns on a fat-tree topology of 8,192-hosts.

k	Hosts	Large flows	Runtime (ms)
16	1024	1024	1.45
16	1024	5000	4.14
32	8192	8192	2.71
32	8192	25000	9.23
32	8192	50000	26.31
48	27648	27648	6.91
48	27648	100000	51.30
48	27648	250000	199.43

Table 4: Demand estimation runtime.

	1,024 hosts		8,192 hosts	
Iterations	$f =$3,215	$f =$6,250	$f =$25k	$f =$50k
1000	2.997	5.042	6.898	11.573
5000	12.209	20.848	19.091	32.079
10000	23.447	40.255	32.912	55.741

Table 5: Runtime (ms) vs. number of Simulated Annealing iterations for different number of flows f.

6.4.3 Complexity of Demand Estimation

Since the demand estimation is performed once per scheduling period, its runtime must be reasonably small so that the length of the control loop is as small as possible. We studied the runtime of demand estimation for different traffic matrices in data centers of varying sizes.

Table 4 shows the runtimes of the demand estimator for different input sizes. The reported runtimes are for runs of the demand estimator using 4 parallel threads of execution on a modest quad-core 2.13 GHz machine. Even for a large data center with 27,648 hosts and 250,000 large flows (average of nearly 10 large flows per host), the runtime of the demand estimation algorithm is only 200 ms. For more common scenarios, the runtime is approximately 50-100ms in our setup. We expect the scheduler machine to be a fairly high performance machine with more cores, thereby still keeping the runtime well under 100ms even for extreme scenarios.

The memory requirement for the demand estimator in our implementation using a sparse matrix representation is less than 20 MB even for the extreme scenario with nearly 250,000 large flows in a data center with 27k hosts. In more common scenarios, with a reasonable number of large flows in the data center, the entire data structure would fit in the L2 cache of a modern CPU.

Considering the simplicity and number of operations involved, an FPGA implementation can store the sparse matrix in an off-chip SRAM. An FPGA such as the Xil-inx Virtex-5 can implement up to 200 parallel processing cores to process this matrix. We estimate that such a configuration would have a computational latency of approximately 5 ms to perform demand estimation even for the case of 250,000 large flows.

6.4.4 Complexity of Simulated Annealing

In Table 5 we show the runtime of Simulated Annealing for different experimental scenarios. The runtime of Simulated Annealing is asymptotically independent of the number of hosts and only dependent on the number of flows. The main takeaway here is the scalability of our Simulated Annealing implementation and its potential for practical application; for networks of thousands of hosts and a reasonable number of flows per host, the Simulated Annealing runtime is on the order of tens of milliseconds, even for 10,000 iterations.

6.4.5 Control Overhead

To evaluate the total control overhead of the centralized scheduling design, we analyzed the overall communication and computation requirements for scheduling. The control loop includes 3 components—all switches in the network send the details of large flows to the scheduler, the scheduler estimates demands of the flows and computes their routes, and the scheduler transmits the new placement of flows to the switches.

We made some assumptions to analyze the length of the control loop. (1) The control plane is made up of 48-port GigE switches with an average 10 μs latency per

	Flows per host		
Hosts	1	5	10
1,024	100.2	100.9	101.7
8,192	101.4	106.8	113.5
27,648	104.6	122.8	145.5

Table 6: Length of control loop (ms).

switch. (2) The format of messages between the switches and the controller are based on the OpenFlow protocol (72B per flow entry) [27]. (3) The total computation time for demand estimation and scheduling of the flows is conservatively assumed to be 100 ms. (4) The last hop link to the scheduler is assumed to be a 10 GigE link. This higher speed last hop link allows a large number of switches to communicate with the scheduler simultaneously. We assumed that the 10 GigE link to the controller can be fully utilized for transfer of scheduling updates.

Table 6 shows the length of the control loop for varying number of large flows per host. The values indicate that the length of the control loop is dominated by the computation time, estimated at 100 ms. These results show the scalability of the centralized scheduling approach for large data centers.

7 Related Work

There has been a recent flood of new research proposals for data center networks; however, none satisfyingly addresses the issue of the network's bisection bandwidth. VL2 [16] and Monsoon [17] propose using per-flow Valiant Load Balancing, which can cause bandwidth losses due to long-term collisions as demonstrated in this work. SEATTLE [25] proposes a single Layer 2 domain with a one-hop switch DHT for MAC address resolution, but does not address multipathing. DCell [19] and BCube [18] suggest using recursively-defined topologies for data center networks, which involves multi-NIC servers and can lead to oversubscribed links with deeper levels. Once again, multipathing is not explicitly addressed.

Researchers have also explored scheduling flows in a multi-path environment from a wide-area context. TeXCP [23] and MATE [10] perform dynamic traffic engineering across multiple paths in the wide-area by using explicit congestion notification packets, which require as yet unavailable switch support. They employ *distributed* traffic engineering, whereas we leverage the data center environment using a tightly-coupled central scheduler. FLARE [31] proposes multipath forwarding in the wide-area on the granularity of *flowlets* (TCP packet bursts); however, it is unclear whether the low intra-data center latencies meet the timing requirements of flowlet bursts to prevent packet reordering and still achieve good per-

formance. Miura *et al.* exploit fat-tree networks by multipathing using tagged-VLANs and commodity PCs [28]. Centralized router control to enforce routing or access control policy has been proposed before by the 4D architecture [15], and projects like Tesseract [35], Ethane [6], and RCP [5], similar in spirit to Hedera's approach to centralized flow scheduling.

Much work has focused on virtual switching fabrics and on individual Clos networks in the abstract, but do not address building an operational multi-level switch architecture using existing commodity components. Turner proposed an optimal non-blocking virtual circuit switch [33], and Smiljanic improved Turner's load balancer and focused on the guarantees the algorithm could provide in a generalized Clos packet-switched network [32]. Oki *et al.* design improved algorithms for scheduling in individual 3-stage Clos switches [30], and Holmburg provides models for simultaneous and incremental scheduling of multi-stage Clos networks [20]. Geoffray and Hoefler describe a number of strategies to increase bisection bandwidth in multistage interconnection networks, specifically focusing on source-routed, per-packet dispersive approaches that break the ordering requirement of TCP/IP over Ethernet [13].

8 Conclusions

The most important finding of our work is that in the pursuit of efficient use of available network resources, a central scheduler with global knowledge of active flows can significantly outperform the state-of-the-art hash-based ECMP load-balancing. We limit the overhead of our approach by focusing our scheduling decisions on the large flows likely responsible for much of the bytes sent across the network. We find that Hedera's performance gains are dependent on the rates and durations of the flows in the network; the benefits are more evident when the network is stressed with many large data transfers both within pods and across the diameter of the network.

In this paper, we have demonstrated the feasibility of building a working prototype of our scheduling system for multi-rooted trees, and have shown that Simulated Annealing almost always outperforms Global First Fit and is capable of delivering near-optimal bisection bandwidth for a wide range of communication patterns both in our physical testbed and in simulated data center networks consisting of thousands of nodes. Given the low computational and latency overheads of our flow placement algorithms, the large investment in network infrastructure associated with data centers (many millions of dollars), and the incremental cost of Hedera's deployment (e.g., one or two servers), we show that dynamic flow scheduling has the potential to deliver substantial bandwidth gains with moderate additional cost.

Acknowledgments

References

[1] Apache Hadoop Project. http://hadoop.apache.org/.

[2] Cisco Data Center Infrastructure 2.5 Design Guide. http://www.cisco.com/univercd/cc/td/doc/solution/dcidg21.pdf.

[3] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of ACM SIGCOMM, 2008*.

[4] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Understanding Data Center Traffic Characteristics. In *Proceedings of ACM WREN, 2009*.

[5] CAESAR, M., CALDWELL, D., FEAMSTER, N., REXFORD, J., SHAIKH, A., AND VAN DER MERWE, J. Design and Implementation of a Routing Control Platform. In *Proceedings of NSDI, 2005*.

[6] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., AND SHENKER, S. Ethane: Taking Control of the Enterprise. In *Proceedings of ACM SIGCOMM, 2007*.

[7] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of OSDI, 2006*.

[8] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI, 2004*.

[9] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of SOSP, 2007*.

[10] ELWALID, A., JIN, C., LOW, S., AND WIDJAJA, I. MATE: MPLS Adaptive Traffic Engineering. *Proceedings of INFOCOM, 2001*.

[11] EVEN, S., ITAI, A., AND SHAMIR, A. On the Complexity of Timetable and Multicommodity Flow Problems. *SIAM Journal on Computing 5, 4 (1976), 691–703*.

[12] FLEISCHER, M. Simulated Annealing: Past, Present, and Future. In *Proceedings of IEEE WSC, 1995*.

[13] GEOFFRAY, P., AND HOEFLER, T. Adaptive Routing Strategies for Modern High Performance Networks. In *Proceedings of IEEE HOTI, 2008*.

[14] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proceedings of SOSP, 2003*.

[15] GREENBERG, A., HJALMTYSSON, G., MALTZ, D. A., MYERS, A., REXFORD, J., XIE, G., YAN, H., ZHAN, J., AND ZHANG, H. A Clean Slate 4D Approach to Network Control and Management. *ACM SIGCOMM CCR, 2005*.

[16] GREENBERG, A., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D., PATEL, P., AND SENGUPTA, S. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of ACM SIGCOMM, 2009*.

[17] GREENBERG, A., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. Towards a Next Generation Data Center Architecture: Scalability and Commoditization. In *Proceedings of ACM PRESTO, 2008*.

[18] GUO, C., LU, G., LI, D., WU, H., ZHANG, X., SHI, Y., TIAN, C., ZHANG, Y., AND LU, S. BCube: A High Performance, Server-Centric Network Architecture for Modular Data Centers. In *Proceedings of ACM SIGCOMM, 2009*.

[19] GUO, C., WU, H., TAN, K., SHI, L., ZHANG, Y., AND LU, S. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. In *Proceedings of ACM SIGCOMM, 2008*.

[20] HOLMBERG, K. Optimization Models for Routing in Switching Networks of Clos Type with Many Stages. *Advanced Modeling and Optimization 10, 1 (2008)*.

[21] HOPPS, C. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, IETF, 2000.

[22] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of ACM EuroSys, 2007*.

[23] KANDULA, S., KATABI, D., DAVIE, B., AND CHARNY, A. Walking the Tightrope: Responsive yet Stable Traffic Engineering. In *Proceedings of ACM SIGCOMM, 2005*.

[24] KANDULA, S., SENGUPTA, S., GREENBERG, A., PATEL, P., AND CHAIKEN, R. The Nature of Data Center Traffic: Measurements & Analysis. In *Proceedings ACM IMC 2009*.

[25] KIM, C., CAESAR, M., AND REXFORD, J. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *Proceedings of ACM SIGCOMM, 2008*.

[26] LOCKWOOD, J. W., MCKEOWN, N., WATSON, G., GIBB, G., HARTKE, P., NAOUS, J., RAGHURAMAN, R., AND LUO, J. NetFPGA–An Open Platform for Gigabit-rate Network Switching and Routing. In *Proceedings of IEEE MSE, 2007*.

[27] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR, 2008*.

[28] MIURA, S., BOKU, T., OKAMOTO, T., AND HANAWA, T. A Dynamic Routing Control System for High-Performance PC Cluster with Multi-path Ethernet Connection. In *Proceedings of IEEE IPDPS, 2008*.

[29] MYSORE, R. N., PAMPORIS, A., FARRINGTON, N., HUANG, N., MIRI, P., RADHAKRISHNAN, S., SUBRAMANYA, V., AND VAHDAT, A. PortLand: A Scalable, Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proceedings of ACM SIGCOMM, 2009*.

[30] OKI, E., JING, Z., ROJAS-CESSA, R., AND CHAO, H. J. Concurrent Round-robin-based Dispatching Schemes for Clos-network Switches. *IEEE/ACM TON, 2002*.

[31] SINHA, S., KANDULA, S., AND KATABI, D. Harnessing TCPs Burstiness using Flowlet Switching. In *Proceedings of ACM HotNets, 2004*.

[32] SMILJANIC, A. Rate and Delay Guarantees Provided by Clos Packet Switches with Load Balancing. *Proceedings of IEEE/ACM TON, 2008*.

[33] TURNER, J. S. An Optimal Nonblocking Multicast Virtual Circuit Switch. In *Proceedings of IEEE INFOCOM, 1994*.

[34] VALIANT, L. G., AND BREBNER, G. J. Universal Schemes for Parallel Communication. In *Proceedings of ACM STOC, 1981*.

[35] YAN, H., MALTZ, D. A., NG, T. S. E., GOGINENI, H., ZHANG, H., AND CAI, Z. Tesseract: A 4D Network Control Plane. In *Proceedings of NSDI, 2007*.