Increasing Datacenter Network Utilisation with GRIN

Alexandru Agache, Razvan Deaconescu, and Costin Raiciu,
University Politehnica of Bucharest

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/agache

This paper is included in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

Open Access to the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15) is sponsored by USENIX

Increasing Datacenter Network Utilisation with GRIN

Alexandru Agache, Razvan Deaconescu and Costin Raiciu
University Politehnica of Bucharest

Abstract

Various full bisection designs have been proposed for datacenter networks. They are provisioned for the worst case in which every server sends flat out and there is no congestion anywhere in the network. However, these topologies are prone to considerable underutilisation in the average case encountered in practice. To utilise spare capacity we propose GRIN, a simple, cheap and easily deployable solution that simply wires up any free ports datacenter servers may have. GRIN allows each server to use up to a maximum amount of bandwidth dependent on the number of available ports and the distribution of idle uplinks in the network. Our evaluation found significant benefits for bandwidth-hungry applications running over our testbed, as well as on 1000 EC2 instances. GRIN can be used to augment any existing datacenter network, with a small initial effort and no additional maintenance costs.

1 Introduction

Datacenter networks are provisioned for peak load, as operators want performance guarantees even when the network is highly utilised. At the extreme, operators provision their network to fully support any possible traffic pattern: the network core is guaranteed never to become a bottleneck, regardless of the traffic patterns generated by the servers; such networks are said to provide *full-bisection bandwidth*. FatTree [3] and VL2 [9] are full-bisection datacenter topologies deployed in production networks. A high profile example is Amazon's EC2 cloud that was using a topology resembling VL2 for their regular instances until recently (see Section 2 in [18]), and are now deploying 10Gbps FatTrees.[1]

Measurement studies show that datacenter networks are underutilised: the traffic has on-off patterns leading many links to run hot for certain periods of time, while even more links are idle, leaving the network core underutilised most of the time [9, 13, 6]. Datacenters heavily rely on the concept of resource pooling: different applications' workloads are multiplexed onto the hardware, and any application can in principle expand to utilise as many resources as it needs as long as there is any available capacity left. Resource pooling does not apply to datacenter networks: although the core is underutilised, hosts cannot take advantage because they are often bottlenecked by their NICs.

[1] Private conversation with Amazon engineers.

In this paper we set out to make datacenter networks better at resource pooling, which would increase their average utilisation and provide better performance per dollar. The obvious solution is to use bigger links between servers and Top-of-Rack switches: in a 10Gbps full-bisection network this means either 40Gbps links or using multiple 10Gbps links (i.e. multihoming). This technique improves performance: when few servers are sending data, they can send many times faster. Unfortunately, it is also very expensive: 40Gbps links are not commodity yet and switches supporting them have small port densities. Multihoming seems more viable economically, but even dual-homing requires doubling the number of ToR switches in the network.

We propose GRIN, a simple change to existing datacenter networks where any free port existing in any *server* is connected to a free port of another server in the same or a neighboring rack. Servers can then communicate using a path provided by the original topology, or via one of the servers they are directly connected to. GRIN can function seamlessly over existing datacenter networks, and the best case benefits are obvious: for every additional port used, a server can send 1 or 10 Gbps more traffic if the neighbour's uplink is idle. As modern NICs are often dual or quad-ports, these additional interfaces should be already available in most servers on the market. [2]

We tested our GRIN implementation with various workloads and applications, using both a small local cluster and on 1000 EC2 hosts. GRIN can significantly increase end-to-end application performance; a 2-port GRIN setup speeds up HDFS, NFS, Spark and Cassandra by a factor of two or more in certain scenarios. While unmodified applications can benefit from GRIN, these gains depend on the traffic pattern of the neighbouring servers. Many datacenter apps run on multiple nodes, and scheduling work across these is already an important component for performance. We have modified a few such apps to be GRIN-aware when taking their scheduling decisions (namely HDFS, Hadoop and Spark [25]). By jointly optimising addressing, routing and applications, GRIN can achieve performance similar to multihoming.

2 Problem Statement

To understand why full-bisection networks are underutilised most of the time, we measure the network utili-

[2] Our brief survey shows most adapters on the market are dual and quad port, and price per port decreases significantly for multi-port NICs.

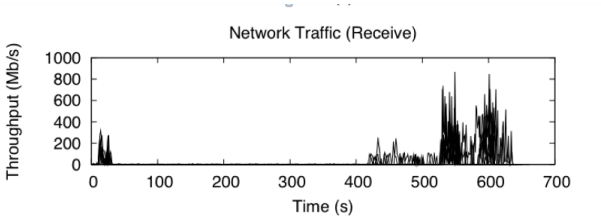


Figure 1: Network utilisation of a simple Map/Reduce job

sation of a small cluster of ten servers connected to non-blocking switch and running a map-reduce job, a typical datacenter application [8]. The servers run Hadoop word-count over a collection of web pages (50GB) and they store at replication level 3. We plot the network throughput measured for one server in Figure 1.

In the map phase, there will be a small percentage of tasks whose data is non-local ([26] estimates 1%-10%), thus requiring filesystem reads from other servers, which will be bottlenecked by the host NIC capacities, assuming appropriate storage provisioning. The shuffle phase will move the data generated by the mappers to the reducers. The shuffle phase is notoriously bandwidth hungry, but this depends on the number of reducers. In the worst case, all servers are reducers and download data at the same time, leading to an all-to-all traffic pattern that fully utilises the network core—in fact, this is the main motivation given for building full-bisection networks [3]. In practice, the number of reducers is an order of magnitude smaller then the number of mappers, and the shuffle phase starts earlier for some servers, thus the core network utilisation will be a lot smaller; still, some reducers will be bottlenecked by their servers' NIC. Finally, the output of the reduce phase is written to disk leading to point-to-point transfers, again bottlenecked by the host NIC.

We want to change existing topologies to allow hosts to utilise as much of the idle parts of the network as possible when other hosts are not active (in the map or output phase of the job above, for example). Good solutions share the following properties:

- **Ability to scale:** cost is a major factor that determines what is feasible to deploy in practice. Using more server ports should increase performance and incur little to no additional costs.
- **Fairness and isolation:** access to neighbour links must be mediated such that each server gets a fair share of the total bandwidth. Misbehaving servers should be penalized, and they should not adversely affect the performance of the network.
- **Widely applicable:** it should be possible to apply the solution to existing or future networks.
- **Incrementally deployable:** it should be possible to deploy the solutions on live datacenter networks with minimal disruption. This implies hardware or software changes to the network core (including routing algorithms) are out-of-scope. Further, upgrading only a subnet should bring appropriate benefits.

Barring extensive changes to the original topology, the most straightforward solution is to multihome servers by using additional TOR switches. We add a TOR switch for every additional server port (see Fig.2b), so that each server is connected to each of the multiple TOR switches from its rack. In order to keep the rest of the topology unchanged, we evenly divide the uplinks of the original TOR switch between all the local TOR switches. The resulting topology is oversubscribed, but now each server can potentially use much more bandwidth. Multihoming brings additional costs in terms of switching equipment, rack-space, energy usage and maintenance. As every additional server port could require an extra switch, this solution does not scale well with the number of server ports.

3 GRIN

Our solution is to interconnect servers directly using additional ports (some of which are already installed, and effectively free), while keeping the original topology unchanged. Each pair of servers that are directly connected in this manner become neighbours. Intuitively, when a server does not need to use its main network interface, it may allow one or more of its neighbours to "borrow" it, by forwarding packets received from them (or packets addressed to them) to their final destination. This solution is depicted in Figure 2c and we call it GRIN.

When a server wishes to transmit, it can use both its uplink and the links leading to its neighbours. Conversely, the destination can be reached through both its uplink and via its neighbours. We call the links used to interconnect servers, horizontal (or GRIN) links, and reserve the term uplinks for those that connect servers to the switch in the original topology. The network interface where the uplink is connected becomes the primary interface of the server, while the others are considered to be secondary interfaces. If every server has n GRIN neighbours, we say that the *degree* of the GRIN topology is equal to n.

Cabling. The baseline GRIN implementation relies on servers being interconnected within the same rack. This is the cheapest solution, and should work at even the highest link speeds. From a performance point of view, it is best to connect those servers that usually do not need to access the network at the same time, otherwise interconnection will not bring major gains beyond improving local throughput. Traditionally, distributed applications tend to localize traffic within racks or pods (multiple racks) because inter-pod bandwidth used to be scarce. In such cases

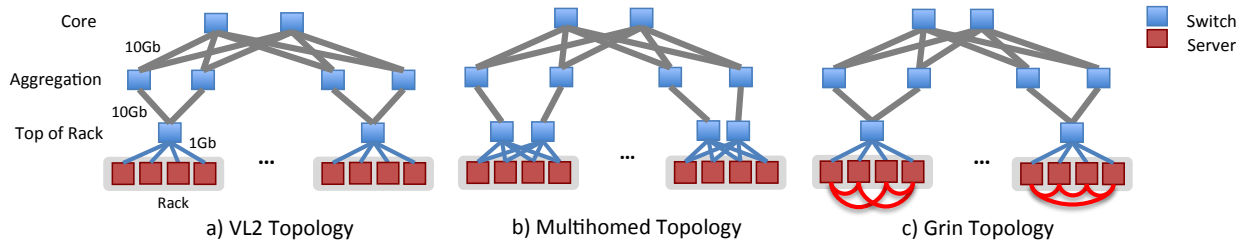


Figure 2: Enhancing a VL2 topology to improve network utilisation

and when nearby servers run the same distributed application, we can enhance the application's scheduler to become GRIN aware (see §3.4). Whenever possible, we can also connect servers from neighbouring racks. Cross-rack cabling is more complex to wire but ensures better fault tolerance to ToR switch failures.

Path Length. Most datacenter topologies have multiple equal cost paths between servers. With GRIN, the set of paths can grow to include elements consisting of increasingly large number of hops. When choosing a path between two random servers, we can select (1) one of the paths available in the original topology, (2) a path that consists entirely of GRIN links, and (3) any number of intermediate servers, and form a path using the concatenation of all intermediate paths.

A compromise must be found between limiting path length and trying to make the most of the available network capacity. One limitation is the fact that any path should traverse at most two uplinks: once in the full-bisection bandwidth network core, there is no reason for traffic to be bounced via another "waypoint" server. Thus, in a GRIN topology a path consists of three segments:

- the first horizontal segment, which is a group of horizontal links going from the source to the server whose uplink is used to reach the first switch
- the path taken from the first to the second uplink through the original network core
- the second horizontal segment, which also consists of a number of GRIN links and goes from the second uplink to the destination server

A horizontal segment can also be empty; for example, any path that was also available in the original topology does not include any horizontal links. We use the term $horizontal_n$ routing (or h_n routing) to describe the fact that in a given GRIN topology we are only interested in paths which have horizontal segments of length at most n; by this definition, the original topology uses h_0 routing.

3.1 A strawman design

For GRIN to work, it needs five key components:

- An algorithm to assign addresses to horizontal GRIN interfaces. We term these *secondary addresses*.
- Techniques to enable servers to discover secondary addresses of other servers, so that they can use them to send traffic.
- Ability to efficiently route traffic to secondary addresses via the appropriate neighbour.
- Ability to create paths spanning multiple horizontal GRIN links.
- A mechanism to spread traffic over the multiple paths

Designing all these seems simple at first sight, and a strawman solution is the following. First, assign datacenter-unique addresses to secondary interfaces in a different subnet from the uplinks to allow servers to distinguish primary and secondary addresses. To enable discovery of secondary addresses, simply use DNS, assuming it is already deployed in the datacenter: register a new "A" record for every secondary interface of server s in the DNS entry for s. Servers discover secondary interfaces by running a DNS lookup.

Routing traffic to a secondary address via a neighbour is trickier. The routing system needs to be informed of the neighbours reachable via a server, and the straightforward solution is to run the datacenter routing protocol (e.g. OSPF) all the way down to the servers, and have neighbour addresses announced in the routing system. This solution is unwieldy: link-state protocols such as OSPF do not scale well beyond a few thousand routers, and pushing servers into OSPF breaks this boundary. Secondly, we need to significantly reconfigure the datacenter routing protocol. Even assuming the routing system can be upgraded, we still have the problem of routing via multiple horizontal links. For this to work, the source must be able to specify a list of intermediary addresses for the packet, and the simplest solution is to use loose source routing. LSR support exists in most operating systems, but it raises performance problems: it does not work well with hardware offloading on some NICs, and LSR packets could even find themselves on the slow path of routers.

One last mechanism is needed to spread traffic over the available paths. As TCP is the de-facto standard transport protocol in datacenter, the straightforward solution is to pin each connection to one path (i.e. round robin), but

there is a danger than a neighbour link is highly congested. In such cases we should move traffic back to the uplink quickly, which is easier said than done—to ensure proper routing we need to change the addresses in the packets which will break the TCP connection. To solve this problem we need to either change the host stack (add some sort of mobility support) or modify the applications.

Towards a solution. The main conclusion arising from the strawman solution above is that dealing with the subproblems in isolation is inefficient because one solution affects many others. For instance, independent addressing forces us to use unscalable routing schemes, and neighbour discovery mechanisms might be redundant if we have a mobility solution in the host stack.

That is why we take a holistic approach to designing these mechanisms, co-optimising them to enable a cheaper and easier to deploy solution. Our first insight is that we could avoid source routing if we limit ourselves to one horizontal hop after the source and one before the destination (h_1 routing). In a h_1 network, the sender can steer outgoing traffic over either the uplink or GRIN links by simply placing them on the appropriate interface. To route via a neighbour of the destination we leverage our addressing scheme (described below). On the other hand, h_1 routing appears to have the least potential of actually using spare network capacity. How much of an improvement, if any, can be achieved by increasing the length of horizontal segments? The results of our evaluation in section 5.1, show that h_1 routing utilises most of the capacity, while being the cheapest from a forwarding point-of-view. That is why we focus on h_1 routing alone in our solution.

The two main building blocks of GRIN, described next, are an addressing scheme that works without changes to the routing system, and using Multipath TCP to efficiently utilise network capacity.

3.2 GRIN addressing

To solve the routing system scalability issues, we relate the assignment of secondary addresses with the addresses of the uplinks, as follows. All network addresses $addr$ are split into two meaningful groups of bits: the most significant 24 bits are the server identifier, $I(addr)$, which must be unique in the datacenter. The last 8 bits represent the GRIN identifier $G(addr)$, which is set to 1 for primary interfaces and larger values for secondary interfaces.

The address of a secondary interface has the same server identifier as the address of the primary interface of the neighbour it connects to; the only difference is the GRIN identifier. Formally, for any server s, let up_s be the address used by its primary interface and n_s be the

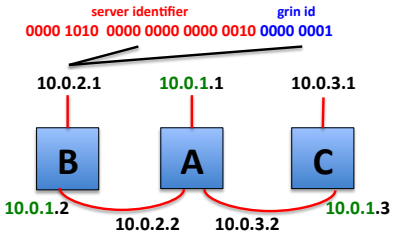


Figure 3: Grin Address Assignment Algorithm

number of neighbours s has connected with already; n_s is zero when we start the address assignment algorithm. To connect server t to server s, the address of the secondary interface is $grin_{t,s}$ and has the following structure:

$$I(grin_{t,s}) = I(up_s) \qquad G(grin_{t,s}) = 2 + n_s$$

Given a secondary address, we can infer the primary address of its neighbour by merely substituting the least significant byte with 1. This scheme also provides an easy way to differentiate between multiple secondary interfaces connected to same neighbour s, because n_s will increase after each subsequent interconnection.

Consider the example in Figure 3, where there are two additional network ports available on each server. Initially we choose to connect servers A and B. Since $up_A = 10.0.1.1$ and $up_B = 10.0.2.1$, we will assign the address 10.0.2.2 to $grin_{A,B}$ and 10.0.1.2 to $grin_{B,A}$ (because both n_A and n_B are 0 initially). Now $n_A = 1$, and $n_C = 0$, so if we interconnect servers A and C $grin_{A,C}$ will be 10.0.3.2, and $grin_{C,A}$ will be 10.0.1.3.

Routing is greatly simplified with this addressing scheme: a simple router configuration change should be enough for most networks to be adapted to GRIN. In fact, many networks already assign subnets instead of individual addresses to servers, to support direct addressing for virtual machines running on them. Finally, the servers will need to be configured to forward traffic they receive for their neighbours. Our addressing scheme sacrifices 8 bits for the GRIN identifier, supporting more than 250 horizontal links. The remaining 24 bits can uniquely identify up to 16 million servers. Fewer bits can be used for GRIN links (e.g. 4) to support a larger number of servers.

3.3 Packet Forwarding

To implement GRIN we can reuse forwarding support provided by modern OSes. Linux, for instance, peaks at a rate of about 570Kpps in our tests. This is good enough for gigabit links, or even at 10 Gbps with jumbo frames, but cannot really keep up as NICs become faster.

Can we do better? When processing packets for its GRIN neighbours, a server is fulfilling three main

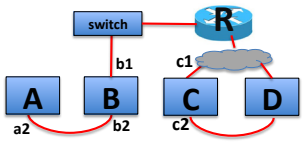


Figure 4: Simple GRIN1 Setup

functions: identification of packets intended for another server, rewriting some header fields and forwarding the result. For packet identification, we can leverage hardware filtering capabilities present in modern NICs which allow packets to be received on different queues based on various discriminants, such as destination address. With IP forwarding, each server must write at least the MAC source and destination addresses in the packet. As no hardware support exists for IP forwarding in commodity NICs, this operation must be performed in software.

In theory, GRIN allows us to avoid this extra work by using *bridging* instead of forwarding. In this context, bridging means simply passing a packet from one interface to another, without doing any kind of software processing on it. The origin server knows the MAC address of the next IP hop for a packet. In an L2 network this is the primary interface of the destination (if the packet is heading for a primary address) or the primary interface of one of the destination's neighbours. For a L3 network, the next hop is the designated first router. In both cases, we use ARP to find the proper MAC address, as the IP address is already known (directly from the packet for L2 and by configuration for L3).

This concept is presented in Figure 4: servers A and B are neighbours, and the default gateway for B's uplink traffic is router R. When A sends a packet to C, with basic forwarding the packet will have the destination IP address $D_{ip}(p) = c_1$ and destination MAC address $D_{mac}(p) = mac(b_2)$. The latter will change over the following hops, first to $mac(r_1)$ and, eventually, $mac(c_1)$. With bridging, the packet leaves A with $D_{ip}(p) = c_1$ and $D_{mac}(p) = mac(r_1)$; it can find $mac(r_1)$ by sending an ARP request. B can simply pass the packet to the uplink upon reception; the original contents are enough to steer it toward C, since the router knows how to reach c_1.

If p is sent to a secondary interface of C, for example c_2, that is connected to a server D, then there is one more significant step to be mentioned. The packet will leave A, and reach R as in the previous case. The routers are configured to send packets with secondary destination addresses to the appropriate neighbour (D for c_2, in this example). Once p reaches server D, it can be placed directly on the link which is connected to c_2, based on the destination IP address. As C receives p, it will have to ignore the incorrect destination MAC address (the last change to this field was made by the router before D). We consider this behaviour to be acceptable, as secondary interfaces only receive packets from their neighbours.

In summary, h_1 routing allows GRIN to not only avoid source routing, but also IP forwarding: intermediate servers can just copy packets between interfaces without "touching" them. Modern commodity 10Gbps NICs (e.g. based on the popular Intel 82599 chipset) already have a simple hardware switch, but it can only move frames between different queues of a single NIC in the *tx*→*rx* scenario; GRIN needs to pass packets from one *rx* queue of an interface to a *tx* queue of another interface (the two interfaces will likely belong to the same multi-port card). With this in place, we could eliminate forwarding overhead altogether. Until hardware support is available, we continue to rely on Linux forwarding for our implementation, described in Section 4, where we also present a prototype that shows the viability of the bridging solution.

3.4 Efficiently using GRIN Topologies

TCP binds a connection to a single NIC, so several parallel TCP connections are needed to utilise the GRIN links available via neighbours. However, according to measurements there are few large ("elephant") flows at any time on a given server [9]; simply adding more NICs should not bring significant performance benefits for most applications over regular TCP. Changing every application to spread data over multiple TCP connections is not feasible: such changes are complex, and there is too much to be changed.

GRIN enables unmodified apps to opportunistically increase their network performance by using Multipath TCP [19]. MPTCP allows a transport connection to grow beyond one NIC (the uplink) and effectively utilise the neighbours' spare capacity. MPTCP also provides dynamic load-balancing across paths: when a path is congested (e.g. via a neighbour) traffic will be automatically shifted to less congested paths [24]. While MPTCP is just one of many possible multi-path forwarding designs, we consider it an ideal enabler for a lightweight implementation, that does not warrant any changes to user applications or the network itself (beyond horizontal links).

Path selection algorithm. Let's see how MPTCP works over a GRIN network. In the example from Fig. 3, assume the only GRIN links are those shown, and that server B wants to send data to server C. The connection begins like regular TCP between the primary interfaces of the two servers. After the initial handshake is complete, server B will be notified via the MPTCP address advertisement mechanism of any additional addresses it can use to con-

tact server C—this mechanism enables neighbour discovery without needing DNS at all.

The set of additional addresses for server C will consist of only one element, $10.0.1.3$. Server B will then attempt to establish a full mesh of subflows between its own addresses (both primary and secondary) and those just received. Thus, we can rely on MPTCP to deal with path selection, and only add small tweaks to the process, as mentioned in Section 4. How soon should we use the additional paths? The easiest answer is to setup additional subflows as soon as address advertisment completes, but this might not always be desirable, especially for short flows—sending few packets over a different subflow raises the probability of a timeout, in case one of the packets gets lost. To protect small flows our implementation uses a configurable threshold, sending all bytes below that via the uplink (100KB for a 1Gbps network); MPTCP will use the other subflows thereafter.

3.5 GRIN-aware applications

A downside of opportunistic usage is the probability than two neighbours will be using their uplinks at the same time; the busier the network is, the higher this probability. However, most datacenter applications have centralized schedulers that decide how to partition the work across the many workers in the system. We can gain performance comparable to multihoming solutions without the associated costs if we modify application schedulers to take into account GRIN links. We have implemented such optimisations for Hadoop, HDFS and Spark [25].

Scatter-gather applications (such as web search) open multiple TCP connections from a frontend server to multiple backend servers. They are bottlenecked by the frontend server's NIC, and susceptible to the "incast" problem [22]. Scatter-gather apps can be easily optimised for GRIN: they just should disable MPTCP and "pin" different TCP flows onto the different available paths for best performance. We have optimised a synthetic scatter-gather frontend server to spread its connections evenly across the GRIN neighbours, thus increasing the total buffer size available to the frontend. We present experimental results for GRIN-aware applications in § 5.6.

4 Implementation

The GRIN implementation works with a MPTCP-enabled Linux kernel and mainly deals with address assignment to secondary interfaces in user-space. We have also made minor changes to the MPTCP kernel in order to improve performance and prevent some unwanted interactions.

The GRIN addressing scheme allows us to use any routing mechanism that was already in place in the original topology, as long as the original addresses can be adapted to the new structure. The assignment itself relies on pre-existing mechanisms, e.g. DHCP.

To automatically configure secondary interfaces, we have implemented a simple server that runs on every computer. It only serves requests that arrive on GRIN interfaces, and its primary functionality is the dissemination of proper secondary addresses to neighbours. After the endpoints of a horizontal link exchange addresses in this manner, each server also adds the required information to the local routing tables. There are two such entries needed per neighbour: one to designate it as the default gateway for all traffic leaving that particular secondary interface, and another to state that the address of the remote endpoint is reachable via the same interface. Additionally, we use Proxy ARP to make servers reply to ARP requests for their neighbours' secondary interfaces, while also making them ignore queries for their own such interfaces.

The first change we made to the MPTCP kernel is related to subflow initiation. Establishing a full mesh, especially for higher GRIN degrees, would setup a very large number of subflows, which is often undesirable. The default behavior for GRIN is to establish the smallest number of subflows such that every horizontal link is used at least once. Thus, in a GRIN topology with degree n, MPTCP will establish n additional subflows. There is also the option of specifying a certain number of subflows, which are going to be selected at random in a manner consistent with the goals of the default behaviour. Another modification was to adjust the MPTCP subflow selection process, which decides what subflow to use when sending each particular packet. In most situations, we want to send data using the direct subflow whenever its congestion window allows it. MPTCP selects the subflow with the smallest RTT when multiple subflows could send a packet, but the RTT estimation alone is noisy and might sent packets on horizontal links even when the uplink is idle. That is why we added bias in favor of the direct connection: the estimated RTT of the uplink is halved for comparison purposes.

Bridged implementation. We have also implemented the prototype of a bridged GRIN1 topology that uses netmap [20] as a stand-in for the missing hardware functionality. Outgoing packets that are heading to a primary interface receive no special treatment. For all others, we make sure that the MAC destination address field contains the L2 address of the proper gateway. For both primary and secondary NICs, we use ethtool to enable ntuple filtering and add filters that make sure any packet destined for the local

server arrives on *rx* queue 0, while all others are received on queue 1. Finally, the netmap bridge ensures that packets received on queue 0 of each NIC are sent to the host TCP stack and packets coming from the host stack are sent using *tx* queue 0. Also, packets received on *rx* queue 1 of any interface are simply sent to *tx* queue 1 of the other interface. This could easily be extended to work with higher GRIN degrees by using an additional *rx/tx* pair of queues for each secondary interface added.

For the setup in Figure 4, when a packet p going from a_2 to c_1 reaches B, it will be placed in *rx* queue 1, because $D_{ip}(p) \neq ip(b_2)$, the netmap bridge will transfer it to *tx* queue 1 of b_1. The switch will direct the packet towards r_1, based on the destination MAC address, and from R it will make its way to the destination. We used the routing implementation in our evaluation because netmap does not support hardware offload when exchanging packets with the Linux TCP stack, affecting performance.

5 Evaluation

This section starts with an analysis of the effect h_n routing can have on GRIN's ability to utilise spare capacity—the result led us to choose h_1 routing and build our solution around it. We continue by evaluating the performance benefits of GRIN, both in synthetic scenarios and for real applications. We also include an assessment of the potential negative impact that GRIN may have in terms of fairness, latency and forwarding overhead. This can be especially important for opportunistic usage.

5.1 How many horizontal hops are needed?

The advantages of h_1 routing in terms of reduced complexity are obvious, but is there anything we lose by using it? We intend to find out if there is any correlation between the maximum allowed path length and the amount of capacity that can be discovered. We model a GRIN topology as racks of computers connected to a single, sufficiently large switch. The computers are interconnected using a variable number of GRIN links (1 to 6), and the entire setup is represented as a directed graph.

We are interested in the maximum network capacity that can be utilised in each case. Given a set of source-destination pairs , we use GLPK [1] to solve the maximum multi-commodity flow problem, which gives the optimal solution and is a hard upper bound on usable capacity. We also solve a couple of specializations to MCF in which we restrict the number of horizontal hops to emulate the best we can do with the corresponding h_n strategy. Due to computational complexity, our network model consisted of six twenty server racks. We run experiments varying the GRIN degree, the maximum number of horizontal hops (h_1, h_2 and no limit), and the traffic pattern:

- *permutation traffic*: each active server sends data to a single destination, and each destination receives data from a single source.
- *group traffic*: servers are randomly assigned to groups such that every group contains the same number of servers. One server is randomly chosen from each group as the destination for the other group members.
- *all-to-all traffic*: each active server sends data to every other active server.
- *random traffic*: the endpoints of every connection are chosen at random.

The results show the total flow is proportional to both the number of active connections and the GRIN degree. There are two interesting observations. First, the difference between h_1 and the optimal solution is at most 32% for GRIN6 and permutation traffic. On average, across all experiments, the difference between h_1 and optimal is just 7%. Also, the difference between the optimal solution and h_2 is seldom more than 1%.

Unfortunately, optimal placement of flows to paths is impossible to achieve, so it's quite hard to form an expectation of real-world behavior based on these results. Any MPTCP connection will only use a limited number subflows in order to prevent performance degradation [24]. Also, it's not usually possible to make informed decisions about flow placement in real time; instead flows are spread over multiple random paths and congestion control balances traffic to get the most out of the network.

To capture these effects, we devised another performance estimation procedure: for a given network and traffic matrix, we start by building the complete set of paths between any source and destination. The length of a path is defined as the number of horizontal segments it contains, with one exception: if a path is made of a single horizontal segment, then its length is zero. The shortest path that goes through the switch is called the direct path. For each connection, we randomly select up to 16 paths (a relatively large number) without replacement and add them to the chosen path set; the direct path is always included. We try to assign the largest possible flow to each element of this set, in ascending order of length. This is done by finding the path segment with the least amount of available capacity, and then using that value to fill the entire path. We applied this method to the same input data as before. We also considered h_3 routing, as without optimal placement, h_2 may no longer be sufficient.

The results are surprising: h_1 provides better results in

%	GRIN degree 1	2	3	4	5	6
10	23/23 21/21	30/30 29/29	39/40 37/33	49/52 39/37	55/65 41/39	61/70 43/42
20	41/41 38/38	56/56 50/49	65/70 61/56	80/90 64/60	92/108 66/63	101/118 68/66
30	57/57 52/52	71/71 65/65	85/92 77/72	103/116 81/77	114/136 83/80	118/143 85/84
40	65/65 63/63	88/88 78/78	106/114 91/85	112/131 93/89	131/153 95/93	136/160 97/96
50	75/75 73/73	100/101 88/89	120/128 100/95	122/139 103/99	135/159 105/103	140/163 107/105
60	87/87 83/83	110/110 97/99	119/134 107/103	129/154 109/107	137/169 112/110	141/174 114/113
70	95/95 92/92	121/122 103/106	122/134 112/110	131/158 114/113	140/170 118/117	149/192 120/120

Figure 5: Hop Analysis Results for Permutation Traffic

most experiments, and the increase is larger as the GRIN degree increases. This behaviour reflects the fact that h_2 and h_3 routing increase contention on horizontal links which leads to more collisions. In the MCF analysis, this effect was offset by the very large number of paths used and exhaustive search used by h_2 and h_3.

Fig. 5 shows detailed evaluation results for permutation traffic. The leftmost column represents the percentage of active servers from each experiment. The first row of each cell shows the maximum flow for h_1 and h_2, respectively, as computed by MCF. The second row has values obtained using our alternative evaluation procedure (h_1 / h_2). We focus on permutation traffic as it exhibits the largest differences in both cases (worst relative behaviour for h_1).

The maximum flow is not included because it is very well approximated by h_2. Note that the total flow values can be larger than 120 (the total number of uplinks) because some connections can be established over entirely horizontal paths. The largest differences between h_1 and h_2 on the first row appear for GRIN degrees unlikely to be encountered in practice. On the second, there are only a few instances where h_2 is marginally better. We conclude that h_1 is the best solution given our constraints.

For other traffic patterns, the differences between h_2 and h_1 are smaller than with MCF, and h_1 gives better performance when measured with our alternative method.

5.2 Experimental setup

We deployed our implementation on a small local cluster of ten servers directly connected to a switch to examine real-world application performance. Each server has a Xeon E5645 processor, 16 GB of RAM, a quad-port gigabit NIC (one port is used for management) and a dual-port 10Gbps NIC. In our testbed, we can build 1Gbps GRIN1 and GRIN2 topologies and a 10Gbps GRIN1 topology.

We use both gigabit and ten gigabit networks in our evaluation. Gigabit links to servers are still in wide use today, and GRIN can offer an immediate and much needed increase in performance for deployed networks assuming extra server ports are available. Our ten gigabit tests aim to establish is GRIN is also applicable to newer networks that use 10Gbps links to the servers.

The small size of the testbed prevented us from building a useful multihomed setup; even if the bandwidth constraints could be enforced, we could only have at most two racks of five servers each which would allow communication at double speed with half of the servers in the testbed. An upper bound is provided instead by simply doubling the results obtained in the original setup.

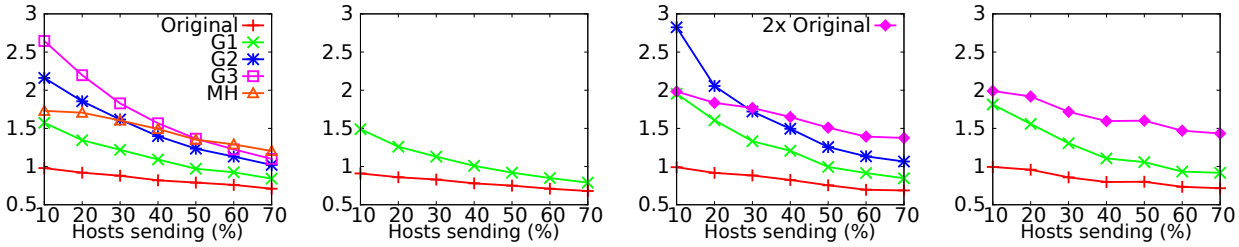
We also deployed a larger GRIN1 topology on 1000 Amazon EC2 c3.large instances that allows us get an impression of our solution's scalability in practice. The instances ran in an Amazon VPC (Virtual Private Cloud), which offers the illusion of an L2 network, with small adjustments to the implementation as proxy ARP did not work in this setting. When dealing with L2 address resolution for secondary interfaces in the kernel, we instead issue an ARP request for the primary address of the corresponding neighbour (which can be easily computed based on the GRIN addressing scheme).

By default, each instance comes with one ENI (elastic network interface), but more (up to three for c3.large) can be added. The first is used for management purposes, the second is the considered the uplink, and the last plays the role of secondary interface. However, unlike regular NICs, all these share the same physical link. We employ dummynet [7] to add bandwidth limitations adding up to less than the maximum available for a single instance, in order to achieve virtual separation. The limit for each interface is set to 100Mbps. At higher speeds, our dummynet configuration led to erratic behaviour.

Finally, to understand the basic properties of GRIN across a wider range of parameters than feasible in practice, we used simulation in *htsim*, a scalable packet level simulator. This has the advantage of giving very precise results and allows us to study reasonably large networks, but doesn't account for factors outside of the transport protocol itself and cannot be used to evaluate applications. Our simulations were based on the same 120 server topology described in the previous section. Increasing network sizes up to tenfold provides qualitatively similar results, however it takes substantially longer to run.

5.3 Basic performance

To understand how GRIN works in practice, we begin our tests with synthetic traffic patterns that we can easily reason about. We use the same patterns described our hop-count evaluation, namely permutation, random, group and all-to-all, and run the experiments in both simulation, on



(a) Simulation, 120 servers (b) EC2, 1000 servers (c) Testbed 1Gbps network (d) Testbed 10Gbps network

Figure 6: GRIN improves performance by 50% to 150% for Random traffic.

Amazon EC2 on 1000 servers and on our local testbed with gigabit and 10 gigabit networks. In all cases we run at least the baseline and GRIN1. We also simulate multihoming, and GRIN3, which we view as an upper bound of the number of ports that GRIN may use in practice.

In Figure 6 we present average per-server throughput results for random traffic as we vary the percent of active servers from 10% to 70%. One thing to keep in mind is that, due to the small size of our local testbed, connections between neighbours will happen more often, and the results will appear better overall. To allow easy comparison between graphs, we normalize the resulting throughput measurements with respect to the best outcome in the original topology (941Mbps for the 1Gbps network, 9960Mbps for 10Gbps and 100Mbps for EC2). Simulation results are normalized by default.

The results show that, as expected, lower percentages of active servers lead to significantly better results for GRIN topologies. Performance improvements are smaller as more servers become active, GRIN2 and GRIN3 performance is better than multihoming until 40% of servers are active, and match it after that. Note that the simulation results are matched very well by EC2 results and are qualitatively similar to the testbed results, giving us confidence in our evaluation. The EC2 results underline the scalability of our solution: a real-life deployment of GRIN1 can run on 1000 servers. Running GRIN at 10Gbps is also worthwhile, doubling the throughput when few servers are active. The results for permutation traffic are similar; the interested reader can refer to [2] for details.

We next turn to all-to-all traffic, a pattern mimicking the shuffle phase of map-reduce. When running this experiment on EC2 we ensure that no server initiates or receives more than 20 concurrent connections to reduce the effects of incast. The results in Figure 7 show that every additional port used with GRIN brings close to 100% performance improvement when few hosts are active. Multihoming is almost always dominated by GRIN2 and GRIN3, however it outperforms GRIN1. As expected, the testbed results are better. The EC2 results accurately track those obtained in simulation.

Finally, the group connection matrix simulates scatter-gather communication. This is the most favorable situation for GRIN topologies, because the large number of sources will fill every link of the receiving server. The results are consistent across both simulation and actual implementation: we get close to the optimal throughput.

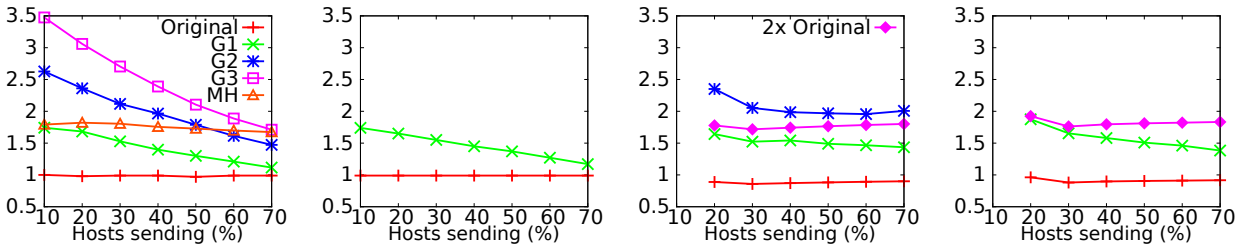
Short flows. We also wanted to find out when GRIN starts to offer benefits if we have fixed-size transfers, assuming there is no contention anywhere in the network. We ran a series of tests using a simple client-server program which requests and then receives a certain number of bytes. The results for the 1Gbps network are shown in Figure 8, and reveal that we need to transmit data on the order hundreds of kilobytes before any sizeable gain becomes apparent for a single connection. That is why we have set the multipath threshold to 100KB in our implementation for this scenario. Also, for smaller transfers (between 15-75K), GRIN may add at most 200μs to the completion time. This issue is caused by the way data is distributed among subflows. At 10Gbps, the threshold increases to 20MB.

5.4 Opportunistic Usage

GRIN can be used opportunistically by simply deploying it and running applications. We deployed a number of real-world applications on a 1Gbps network, where GRIN can have an immediate impact.

The first application is an **NFS** [16] server. Our goal was to measure the time it took to read every file from an exported directory. We varied the file size from one experiment to another, while ensuring their aggregate size was 2GB. As can be seen in Figure 9, throughput improvement is directly proportional to file size. After a certain point, each file is large enough to make the request overhead almost negligible in relation to the actual transfer duration.

Another application we considered is **HDFS**[21]. This is a natural choice for any GRIN setup because it involves handling large amounts of data, so we potentially have a lot to gain in terms of performance. One server was used to host the NameNode, 8 were running DataNodes,



(a) Simulation, 120 servers (b) EC2, 1000 servers (c) Testbed 1Gbps network (d) Testbed 10Gbps network

Figure 7: GRIN improves performance by 80% to 250% in the All-to-all traffic pattern.

and the last one acted as a client. We measured the time needed to transfer a 4GB file from HDFS to local storage. With GRIN1 we got the best possible result, as the file transferred almost twice as fast. The switch to GRIN2 however, did not bring the threefold increase in speed we were hoping for. This was apparently caused by the java process becoming CPU bound. When we requested two transfers in parallel, each process ran on a different core and every network interface was fully utilised.

Next, we wanted to see if GRIN can bring any benefits to **virtual machine migration**. We installed the Xen Hypervisor[5] version 4.2 on several servers running our modified version of the MPTCP kernel. The virtual machine created for the experiments had 4GB of RAM, and its disk image was shared by an network block device server. The metric used during each test was the time required to migrate the VM to another server. Our first attempts, using the xl toolstack, met with failure because it uses ssh to send migration data, which incurs a hefty overhead, so the network is no longer a bottleneck. We got the best results by switching to xm, which uses plain TCP. In this case, using GRIN1 increased migration speed by around 60%, while GRIN2 doubled it.

The last application in our 1Gbps test suite was **Apache Cassandra**[14]. We started a Cassandra cluster consisting of 9 servers, while the last one acted as client. Our goal was to use the Cassandra-stress tool to measure the time required to write a constant amount of data in different circumstances. We used the default values for most parameter, only changing the number of columns to 10, and then varying the size of each column and the numbers of keys inserted such that the total transfer size was around 2GB. The relation between column size and request completion time is presented in Figure 10. Somewhat unsurprisingly, the way parameters combine to determine the amount of data per row is what matters most in terms of performance. The operations complete much more quickly for a smaller number of larger rows. Increasing the number of columns while decreasing the number of rows will also lead to better results, but not to the same extent as using larger rather than more columns.

10Gbps networks. Regular applications don't scale nearly as well as the iperf experiments do at 10Gbps, even with jumbo frames and hardware offload functionality enabled. Thus, we focused on a few apps have high bandwidth requirements and are fast enough to take advantage of it: NFS, HDFS and Spark [25].

With NFS, the server hosts a single 12 GB file which can be transferred by one client in around 10 seconds with GRIN disabled, and a little less than that with one secondary interface enabled. GRIN does not help because the client is CPU-bound. If two clients attempt to transfer the same file *simultaneously* over a GRIN1 network, they both finish in around 10.5 seconds, implying that the server was able to fully utilise both its 10Gbps interfaces.

We used a variable number of Spark workers, connected to a single-node DFS deployment, to count the occurrences of a string in the same 12GB file. At first, GRIN is disabled. A single worker completes the task in 14.5 seconds on average. We need two workers to reduce to time to 10.5 seconds, which is very close to the duration of the data transfer, so we can consider the computation to be finally network-bound. With GRIN1 enabled for both workers, the completion time drops to 7.5 seconds. By starting a third one we can improve this result to around 5.5 seconds, and Spark is now network-limited.

5.5 Perils of Opportunistic GRIN Usage

There are a number of issues that may be caused by the transition to a GRIN topology. The additional flows may increase buffer pressure and overall latency, while servers could find themselves competing with neighbours for their own uplinks. Our main goal in this regard is to do no worse than the original topology. To deal with these two issues we employ a simple priority scheme, based on DSCP. Each direct flow receives a high-priority code point (such as EF), while secondary flows retain the default low value. We rely on iperf to test the fair use of uplinks, and on a simple client-server program to measure the latency of small transfers. GRIN specific contention may happen in two distinct situations: server-local when multiple flows use the uplink and at the switch, on the egress

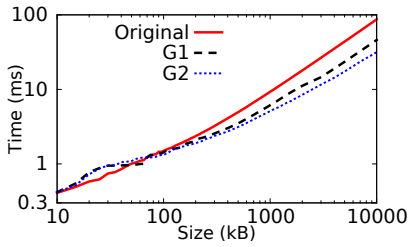


Figure 8: Improvements depend on the size of the transfer.

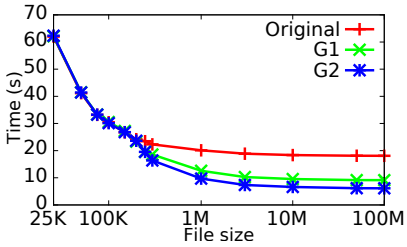


Figure 9: NFS

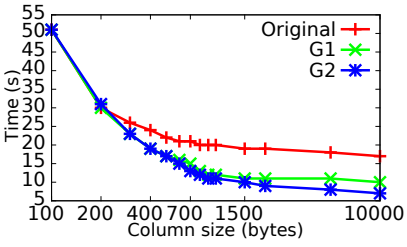


Figure 10: Cassandra

port leading to a particular server. In order to honor DSCP markings locally, servers use a priority-aware queuing discipline, such as PRIO in Linux. Our experiments show that a single, high-priority flow is able to fully utilise the uplink, regardless of the presence of low priority flows.

An idle 10Gbps link takes 100μs on average to transfer 1KB, and 140μs for 100KB with TCP. When competing with several running iperf connections without priorities, the transfer time increases to around 1.8ms in both situations. With the PRIO qdisc, it decreases to 350μs for 1KB and 400μs for 100K. On the downlink, the transfer time grows in both cases to around 2.4ms without priority. If we add priority and configure the switch to discriminate based on DSCP marking, the latency drops to around 110μs and 160μs, respectively.

Does enabling GRIN slow down resource-intensive local applications? Our tests show that GRIN forwarding does not impact storage bound apps, but it is interesting to examine CPU bound and memory bound scenarios.

We ran three resource-intensive applications, Linux kernel compilation, video transcoding and a memcached server, on one of our 6-core Xeon servers. Where relevant, we use a ram disk for persistent storage. Running time is the metric for kernel compilation and transcoding. For memcached, we preload 2^{20} keys with corresponding 60 byte values, and then measure the number of requests that can be fulfilled during 20 second intervals. The requests are generated by 60 local threads that connect to the server using UNIX sockets.

The first two lines from the results in Figure 11 show that running the app on five cores gives near-identical performance regardless of whether the sixth core is idle or forwarding traffic bidirectionally at 10Gbps: if we can spare a core for forwarding, there will be negligible impact on all other apps. The last two lines show the overhead when we run the app on all cores: here forwarding decreases application performance by 9%-13%.

We stress, however, that in many cases clusters of computers are dedicated to one distributed application (e.g. web-search) to avoid bad performance interactions. In such cases the side-effects of forwarding are irrelevant as long as the application as a whole runs faster. In the next section we describe results with GRIN-aware applications where the total completion time is reduced despite the negative effect of forwarding.

5.6 GRIN-aware applications

One simple and very effective optimisation is to schedule bandwidth intensive jobs onto servers that are not direct GRIN neighbors. We have optimised HDFS for reads by placing replicas of the same block in this manner. When a read request comes in, the scheduler replies with the least-recently accessed server that has a replica, and records that the server and his neighbor were "accessed".

We deployed the optimised version of HDFS on 1000 EC2 instances. In each experiment we used a fraction of the nodes to transfer a 400MB file from HDFS to local storage. We used a replication factor of three and a block size equal to 140MB. The results, found in Figure 12, compare the default implementation of HDFS, HDFS running over GRIN1 and, finally, HDFS optimised for GRIN. The results show that running HDFS opportunistically improves download time on average by 14%, and running optimised HDFS brings a 28% improvement. The results also show that, as expected, the optimised version is superior at higher loads, where the probability of GRIN neighbours to be active is much higher.

Next, we optimised Hadoop and Spark by changing the job placement algorithm to avoid scheduling mappers and reducers on neighbour nodes, whenever possible. We ran Spark to run the same string occurrence problem in a large 24GB file over the 1Gbps network. Two HDFS nodes store the data and two Spark nodes do the actual processing. Without GRIN, the total execution time is around 111 seconds. Even if all nodes are grouped up together, using GRIN1 brings the execution time down to 106 seconds. This is caused by a somewhat uneven distribution of the data (one node holds 11G and the other 13G). Since the application can process it pretty fast, GRIN allows one server to help the other out after it finishes sending all the local data. With optimisations enabled, every node has an idle neighbour, and the execution time drops

	kernel	memcache	transcode
idle, 5 cores	137s	5.9 mreq/s	122s
fw, 5 cores	139s	5.9 mreq/s	124s
idle, 6 cores	118s	6m mreq/s	106s
fw, 6 cores	132s	5.5 mreq/s	120s

Figure 11: GRIN forwarding brings little overheads if resource-hungry apps run on separate cores from forwarding. In the worst case, the overhead is 13%

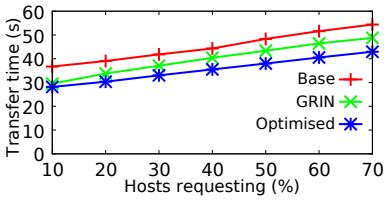


Figure 12: HDFS running on 1000 EC2 instances

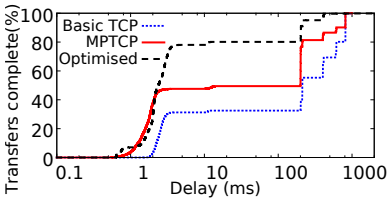


Figure 13: Optimising scatter-gather apps for GRIN

to around 54.5s. A similar optimisation for Hadoop reduces the overall shuffle duration by 20%.

A GRIN topology could also be used to lessen the effects of incast by leveraging the additional switch buffers available at neighbors. In this case, it makes no sense to spread data over multiple paths with MPTCP.

Instead, the frontend can disable MPTCP and split its connections over its uplink and GRIN interfaces in a round-robin fashion. To test this optimisation, we used synthetic scatter-gather application to periodically request data from multiple sources. Figure 13 shows the behavior encountered when one server simultaneously requests 30KB of data over 27 persistent connections, evenly distributed among the nine remaining servers. There are 50 rounds of transfers, each being followed by a 300ms waiting period. Here we compare the original topology with a GRIN2 setup using MPTCP and another one using the incast optimisation. For each case, we plot the CDF of transfer times. Incast mode provides dramatic improvements, reducing the mean by two orders of magnitude.

6 Related Work

A preliminary version of this paper has appeared as [2]. Various solutions have been proposed for increasing core utilisation in full bisection networks, such as MPTCP [17] or Hedera [4]. These, however, aim to make full-bisection topologies behave like a non-blocking switch by routing flows in the network to avoid collisions.

Other approaches, such as Flyways [12] or C-Through [23], are based on augmenting an oversubscribed network with additional communication channels that can be used to improve throughput between different groups of servers when the initial latency is not an issue. They try to create the illusion of full bisection in oversubscribed networks; however, the network activity of a single server is still limited by its uplink.

Using datacenter servers to forward traffic is not a new idea. In fact, topologies such as DCell[11] or BCube[10] rely on servers having multiple ports, and most forwarding is done by the servers themselves. However, these proposals are very difficult to wire, involve complex routing schemes and impose a great forwarding effort on servers. These drawbacks have prevented adoption in practice. GRIN borrows the idea of server routing but uses it in a very simple setup where wiring and routing are trivial, while inherently limiting the amount of forwarded traffic.

Proposals such as ServerSwitch [15] show how forwarding can be implemented in the NIC, without involving the host; such proposals would prevent GRIN forwarding from interfering with host applications.

7 Conclusions

As long as the network is the main bottleneck, wiring up free server ports with GRIN is a simple yet powerful solution to increase the amount of bandwidth available to end-hosts. It is cheap and feasible to implement over almost any topology used today, and this can be done in an incremental fashion. While full-bisection networks show the most potential, we believe it can also be used with oversubscribed networks as long as there still is significant underutilisation.

Even when the additional network ports are not “free”, GRIN can offer an interesting trade-off where we get more capacity out of the network by investing at the edge. There is also the possibility of trading additional cabling costs and complexity to alleviate ToR-level congestion. The server forwarding overhead can be considered an issue in certain opportunistic usage scenarios, but we argue that it can be eliminated altogether with proper hardware support. Moreover, making distributed applications GRIN-aware can significantly diminish any possible shortcoming of the setup. GRIN offers better peak performance, is cheaper, and scales better than alternative solutions like multihoming.

Acknowledgements

This work was supported by Trilogy 2, a research project funded by the European Commission in its Seventh Framework program (FP7 317756). We would like to thank the anonymous reviewers for their feedback and our shepherd George Porter for his help in revising the paper.

References

[1] GNU Linear Programming Kit. `http://www.gnu.org/software/glpk/`.

[2] A. Agache and C. Raiciu. Grin: utilizing the empty half of full bisection networks. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

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

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

[5] P. Barham, B. Dragovic, K. Fraser, S. H, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *In SOSP (2003*, pages 164–177.

[6] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, Jan. 2010.

[7] M. Carbone and L. Rizzo. Dummynet revisited. *SIGCOMM Comput. Commun. Rev.*, 40(2):12–20, Apr. 2010.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[9] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of ACM SIGCOMM 2009*.

[10] 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 *Proceedings of ACM SIGCOMM 2009*.

[11] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 75–86, New York, NY, USA, 2008. ACM.

[12] D. Halperin, S. Kandula, J. Padhye, P. Bahl and D. Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *Proceedings of ACM SIGCOMM 2011*.

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

[14] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[15] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. Serverswitch: a programmable and high performance platform for data center networks. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[16] B. Pawlowski, D. Noveck, D. Robinson, and R. Thurlow. The nfs version 4 protocol. In *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000*, 2000.

[17] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath TCP. In *Proceedings of ACM SIGCOMM 2011*.

[18] C. Raiciu, M. Ionescu, and D. Niculescu. Opening up black box networks with CloudTalk. In *Proceedings of USENIX HotCloud 2012*.

[19] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[20] L. Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of USENIX ATC 2012*.

[21] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[22] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. *SIGCOMM Comput. Commun. Rev.*, 39(4):303–314, Aug. 2009.

[23] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiãnnaki, T. E. Ng, M. Kozuch, and M. Ryan. c-Through: Part-time optics in data centers. In *Proceedings of ACM SIGCOMM 2010*.

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

[25] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[26] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.