Minimal Rewiring: Efficient Live Expansion for Clos Data Center Networks

Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, and Amin Vahdat, *Google, Inc.*

https://www.usenix.org/conference/nsdi19/presentation/zhao

This paper is included in the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).

February 26–28, 2019 • Boston, MA, USA

ISBN 978-1-931971-49-2

Open access to the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19) is sponsored by

NetApp®

Minimal Rewiring: Efficient Live Expansion for Clos Data Center Networks

Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, Amin Vahdat
Google, Inc.

Abstract

Clos topologies have been widely adopted for large-scale data center networks (DCNs), but it has been difficult to support incremental expansions for Clos DCNs. Some prior work has claimed that the structure of Clos topologies hinders incremental expansion.

We demonstrate that it is indeed possible to design expandable Clos DCNs, and to expand them while they are carrying live traffic, without incurring packet loss. We use a layer of patch panels between blocks of switches in a Clos DCN, which makes physical rewiring feasible, and we describe how to use integer linear programming (ILP) to minimize the number of patch-panel connections that must be changed, which makes expansions faster and cheaper. We also describe a block-aggregation technique that makes our ILP approach scalable. We tested our "minimal-rewiring" solver on two kinds of fine-grained expansions using 2250 synthetic DCN topologies, and found that the solver can handle 99% of these cases while changing under 25% of the connections. Compared to prior approaches, this solver (on average) reduces the average number of "stages" per expansion from 4 to 1.29, and reduces the number of wires changed by an order of magnitude or more – a significant improvement to our operational costs, and to our exposure (during expansions) to capacity-reducing faults.

1 Introduction

Large-scale Cloud and Internet-application providers are building many data centers, which can contain tens of thousands of machines, consuming tens of MW. These need large-scale high-speed data-center networks. Historically, these networks were built all at once, at the time of cluster commissioning. However, these data centers are filled with servers and storage gradually, often taking 1-2 years to reach capacity. This mismatch leaves substantial capacity idle, waiting for workloads to arrive. Idle capacity not only costs money, but also lengthens the technology-refresh cycle, which can decrease usable compute capacity – the latest servers are hobbled if they must use old network technology that lacks modern congestion-control schemes, speed increases, and latency improvements. Hence, we usually start by building a moderate-scale network, and then continually expand the network just ahead of server arrival – *while the network is carrying live traffic*. Live incremental expansion can save millions of dollars in network costs while (more importantly) providing the best possible support for compute and storage infrastructure. However, a naive approach can itself create large, unnecessary costs.

Clos topologies are the *de-facto* standard DCN architecture because they support large-scale DCNs from commodity switches [27, 5, 10, 13]. At Google, our *Jupiter* DCNs are Clos topologies. Different variants of Clos DCN structures, such as Fat Tree [1], VL2 [13], F10 [23], Aspen Tree [30], Rotation Striping [33], etc. have been proposed. However, none of these topologies supports fine-grained incremental expansion. First, some Clos topologies (e.g., Fat Tree) can only be built at certain sizes, which fundamentally prevents incremental expansion. Second, even though some of the Clos topologies (e.g., Rotation Striping) can be constructed at arbitrary sizes, incremental expansion can be expensive, because it requires changing a large fraction of the wiring (see §5.3). In fact, [28] has claimed that the structure of Clos topologies hinders fine-grained incremental expansion; this was an explicit motivation for less-structured topologies that can also exploit commodity switches, such as Jellyfish [28] and Random Folded Clos [4]. In this paper, we show that fine-grained incremental expansion of Clos DCNs is, in fact, feasible, with a novel topology-design methodology.

We want to expand a network *live*: without taking it out of service, which would *strand* compute and storage capacity because those machines would not be usable during expansion, and which would also require us to stop or migrate the applications using that network – a disruptive and expensive process. Live expansion requires maintaining sufficient network throughput during the entire course of a live expansion; to avoid congestion, we must therefore do each expansion in multiple automated stages, each of which only disconnects and adds a limited subset of the network elements. We would also like to complete each stage as quickly as possible, since rewiring does reduce our spare capacity, and thus exposes us to an increased risk of simultaneous failures.

Over the course of a multi-stage expansion, we may need to rewire many links. If we were to directly connect links between switches, the resulting manual labor for moving long wires would be slow, expensive, and error-prone. Instead, we introduce a patch-panel layer in our Clos DCNs (see Fig. 1). These DCNs are three-tier Clos topologies, with tier-1 top-of-rack (ToR) switches connected to tier-2 server blocks, each of which connects to a set of tier-3 spine blocks. By connecting all the server blocks and all the spine blocks through a group of patch-panels, a DCN topology can thus be created and modified by simply moving fiber jumpers on the back side of the patch panels. Each series of rewiring steps can hence be done in proximity to a single patch panel, although an entire stage may require touching several panels.

Our scale has grown to the point where a simple version of this patch-panel-based expansion technique is too slow to support the rate at which we must execute expansions. Therefore, we needed to minimize the number of rewirings per expansion, while maintaining bandwidth guarantees.

The primary contribution of this paper is a minimal-rewiring solver for Clos DCN topology design. In the literature, most Clos DCN topologies are designed purely to optimize cost and/or performance at a single chosen size [1, 13, 33, 23, 30]. In contrast, our solver explicitly considers the pre-existing topology when designing a larger one. Our solver uses Integer Linear Programming (ILP) to directly minimize the total number of rewirings. By enforcing a number of balance-related constraints, the resulting topology is also guaranteed to have high capacity and high failure resiliency. With minimal rewiring, a DCN expansion can be done in fewer stages, while still maintaining high residual bandwidth during expansions.

Because we build each DCN incrementally over a period of years, we need to incorporate new technologies incrementally via expansions, such as higher-radix switches or faster links. Our ILP-based formulation incorporates various heterogeneities, including different physical structures, switch radices, port speeds, etc., inside a single DCN.

ILP is NP-hard in general, and does not scale well for large-scale DCNs. It may take hundreds of thousands of integer decision variables to formulate a large-scale DCN. Even the most advanced commercial solver, Gurobi [17], might run for days without computing a solution. We tested a simple version of our ILP-based solver on 4500 synthesized DCN configurations, and found that the solver failed to solve 68% of the configurations within a 3-hour limit. (Longer timeouts yield little improvement.)

To make our solver scale, we developed a block-aggregation technique to reduce the number of decision variables in the ILP formulation. Block aggregation exploits various homogeneities in a DCN, and aggregates decision variables whenever possible. We have a proof that the aggregated decision variables can be decomposed in a later step [32] . Our block-aggregation technique can use different aggregation strategies. With the fastest strategy, all 4500 synthesized DCN configurations can be solved within 10 seconds.

We measure the quality of our solutions in terms of a *rewiring ratio*, the fraction of wires between server blocks and spine blocks in the pre-existing topology that must be disconnected during an expansion. When we use block aggregation, we face a tradeoff: aggregation improves runtime scalability, but sacrifices rewiring optimality. However, we cannot predict the aggregation strategy that will produce the best (lowest) rewiring ratio subject to a chosen deadline. Therefore, our *parallel solver* runs multiple minimal-rewiring solvers with different aggregation strategies at the same time, and picks the solution with the lowest rewiring ratio. This allows us to solve about 99% of the synthesized DCN configurations with a rewiring ratio under 0.25; the median ratio is under 0.05. In turn, these low rewiring ratios allow us to significantly accelerate the entire expansion pipeline. For example, under a constraint that preserves 70% of the pre-expansion bandwidth during expansion, our minimal-rewiring solver reduces the average number of expansion stages required from 4 to 1.29.

2 Prior Work on Expansions

Prior work has described DCN designs that support incremental expansion, and techniques for conducting expansions. Our work focuses on Clos topologies, the de-facto standard for large-scale DCNs; most prior work on expansions has used non-Clos designs.

DCell [15] and BCube [14] are built using iterative structures. As a result, they can only supports expansions at a very coarse granularity, which could lead to substantial stranded DCN capacity after expansion. Similar iteratively-designed DCN structures are also proposed in [16, 21, 22].

JellyFish [28], Space Shuffle [31], Scafida [18], and Expander [29, 9] were designed to support fine-grained incremental expansion using random-graph-based DCN topologies. However, these topologies have not been widely adopted for industrial-scale data centers, possibly due to the increased complexity of cabling and routing and congestion control when deploying large-scale DCNs.

Random Folded Clos [4] is a variant of Clos that supports fine-grained incremental expansion. It maintains a layered structure, but builds inter-layer links based on random graphs. However, Random Folded Clos is only designed for homogeneous DCNs, where all blocks are of the same size and the same port speed. Further, Random Folded Clos is not non-blocking, with reduced capacity when compared to Fat Trees. In contrast, our minimal-rewiring solver can be applied to heterogeneous DCNs, and its topology solution preserves the non-blocking property of Clos topologies.

Similar to our minimal-rewiring DCN topology solver,

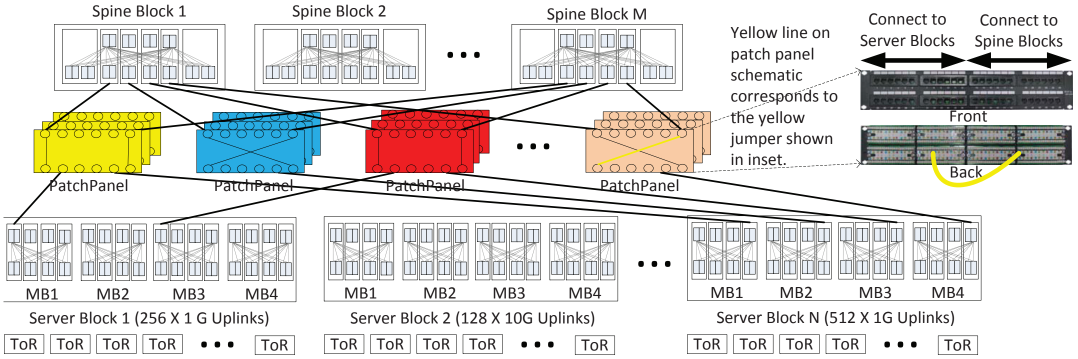

Spine Block 1
Spine Block 2
Spine Block M
• • •
Yellow line on patch panel schematic corresponds to the yellow jumper shown in inset.
Connect to Server Blocks
Connect to Spine Blocks
Front
Back
PatchPanel
PatchPanel
PatchPanel
• • •
PatchPanel
MB1
MB2
MB3
MB4
Server Block 1 (256 X 1 G Uplinks)
MB1
MB2
MB3
MB4
Server Block 2 (128 X 10G Uplinks)
• • •
MB1
MB2
MB3
MB4
Server Block N (512 X 1G Uplinks)
ToR ToR ToR ToR • • • ToR
ToR ToR ToR ToR • • • ToR
ToR ToR ToR ToR • • • ToR

Figure 1: Clos topology with a patch-panel layer, colored to indicate an example 4-stage expansion

optimization-based approaches were also adopted in LEGUP [8] and REWIRE [7]. However, neither paper looked at the topology-design problem in the presence of a patch-panel layer, and could have daunting cabling complexity. Further, LEGUP uses a branch-and-bound algorithm [3], and REWIRE uses simulated annealing [24] to search for the optimal topology. Both algorithms scale poorly, as their convergence time grows exponentially with the problem size.

Condor [25] allowed designers to express goals for DCN topologies via a constraint-based language, and used a constraint solver to generate topologies. Condor addressed the challenge of multi-phase expansions, but their approach was unable to solve expansions for many arbitrary configurations of Clos networks due to computational complexity.

3 Overview of Clos-Based DCN Topology

The Clos topology was initially designed for telephone networks [6]. Years later, it was proposed for DCNs [1] and subsequently became the de-facto standard for large-scale DCNs, at Google [27], Facebook [10], Cisco [5], Microsoft [13], etc.. There are many advantages to Clos networks. First, Clos networks are non-blocking, and thus have high throughput. Second, Clos networks can be built using identical and inexpensive commodity switches, and thus are cost-effective. Third, Clos networks have many of redundant paths for each source-destination pair, and thus are highly failure-resilient.

Clos-based data centers exhibit a layered architecture (see Fig. 1). The lower layer contains a number of *server blocks*[1].The upper layer contains a number of *spine blocks*, used to forward traffic between server blocks. A DCN topology interconnects each server block to each spine block. Because we want to support technology evolution within a single network, each of the server blocks could have a different number of spine-facing uplinks, a different uplink rate, etc[2].

Connecting server blocks and spine blocks by direct wires[3] is highly inefficient. First,a large-scale data center typically has tens of thousands of DCN links. Second, the server blocks and the spine blocks of a data center may be deployed at different locations on a data center floor, due to space constraints, so some direct links would have to run a long way across the data center. Third, moving, adding, or removing a long link during an expansion requires significant human labor, creates a risk of error, and because the new links might have dramatically different lengths requires a large inventory of cables of various lengths.

To overcome these difficulties, we introduce a patch panel [19, 26] layer between server blocks and spine blocks (Fig. 1). Patch panels are much cheaper than other DCN components (e.g., switches, optical transceivers, long-reach fibers). All the interconnecting ports of the server and spine blocks can be connected to the front side of the patch panels via fiber bundles, and all the connecting links can be established or changed using fiber jumpers on the back side[4]. These patch panels are co-located. As a result, a DCN topology can be wired and modified without walking around the data center floor or requiring the addition or removal of existing fiber. Also, as discussed in [27], deploying fibers in bundles greatly reduces cost and complexity; using patch panels means we can deploy bundles once, without having to change them during an expansion. This patch-panel layer makes it much easier for us to support rapid expansions without excessive capacity reduction.

Patch panels allow us to divide a DCN topology into two layers, *physical* and *logical*. As shown in Fig. 1, each server and spine block is connected to the patch-panel layer; we call this the *physical topology*. When a new block is first deployed, we deploy its corresponding physical topology. Changing physical links is not easy, as it involves moving fiber bundles across different patch panels. Hence, in this paper, readers can view physical links as fixed once deployed.

Logical topology defines how server blocks connect to spine blocks, abstracting out the patch-panel layer. All DCN topologies discussed in literatures refer to the logical topology. Many DCN performance metrics have been defined in terms of logical topologies, including network bandwidth,

[1] "Server blocks" are also called "pods" [11], or "edge aggregation blocks" [27].

[2] Fig. 1 shows three server blocks with different uplink configurations.

[3] We use the term "wires" to loosely refer to either fiber or copper links.

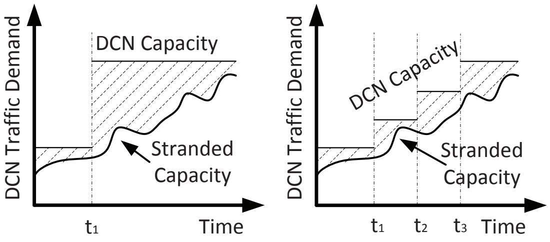
[4] The inset photo in Fig. 1 depicts how we use these jumpers. The yellow line on the right-most patch panel corresponds to the yellow jumper in the inset, which connects one server-block link to one spine-block link.

failure resiliency, incremental expandability, etc. However, except for Condor [25], no prior work has studied the logical-topology design problem in presence of patch panels. As discussed in §5.2.1, in addition to optimizing the performance metrics listed above, we also need to enforce additional physical constraints, so that the resulting logical topology is compatible with the underlying physical topology.

3.1 Considerations for Clos expansions

During an expansion, the existing physical topology remains unchanged, and we only add new bundles of physical links for the newly added/upgraded server/spine blocks. Note that adding new physical links does not impact any ongoing traffic. In contrast, some of the links in the existing logical topology, which could be carrying significant traffic, will have to change, so we must ensure there is no traffic loss caused by changes to the logical topology.

As shown in Fig. 2 (a), we could add a large amount of capacity to a data center during each expansion, which would allow us to do expansions infrequently. However, this expansion strategy would lead to much more *stranded* network capacity – capacity installed but not usable – as the traffic demand could be far less than the capacity provided. By doing fine-grained expansions, we reduce the mean amount of stranded capacity (see Fig. 2 (b)).



(a) Coarse-Grained Expansion. (b) Fine-Grained Expansion.

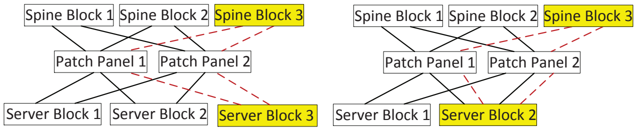
Figure 2: DCN stranded capacity.

4 Patch-Panel Based Expansion Pipeline

We support two types of expansions, at the granularity of a server block (Fig. 3). The first type adds a new server block, to allow more servers to be added to an existing data center. The second type increases the uplink count ("radix") of an existing server block. Typically, the uplinks of a server block are not initially fully populated with optical transceivers, because transceivers are expensive and a new server block has a low bandwidth requirement (as not all of its servers are connected). As a block's bandwidth requirement increases, we need to populate more uplinks. Note that as we expand the server-block layer, additional spine blocks will also be needed, so expansions generally involve adding server blocks and spine blocks at the same time.

Fig. 4 depicts our pipeline for updating a logical topology during a live expansion. It guarantees that:

- No traffic loss due to routing packets into "black holes."



(a) Add a third server block (b) Radix upgrade of a server block

Figure 3: Data center expansion types.

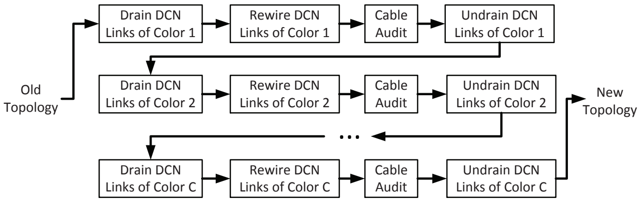

Old Topology
Drain DCN Links of Color 1
Rewire DCN Links of Color 1
Cable Audit
Undrain DCN Links of Color 1
Drain DCN Links of Color 2
Rewire DCN Links of Color 2
Cable Audit
Undrain DCN Links of Color 2
...
Drain DCN Links of Color C
Rewire DCN Links of Color C
Cable Audit
Undrain DCN Links of Color C
New Topology

Figure 4: DCN expansion pipeline.

- All wiring changes are made correctly.
- No congestion due to the temporary capacity reduction.

To avoid black holes when changing a set of logical links, we must redirect traffic on these links to other paths. We instruct our SDN-based control plane to "drain" these links. After we verify that there is no longer traffic on the target drained links, we can proceed to rewire the links.

Rewiring links via patch-panel changes is the most labor-intensive and error-prone step. Typically, thousands of links need to be rewired during one expansion, creating the possibility of multiple physical mistakes during the rewiring process. To check for errors, we perform a *cable audit*. In cable audit, we use the Link Layer Discovery Protocol to construct a post-wiring topology, and then cross-validate this against the target logical topology. We also run a bit-error-rate test (BERT) for all the new links, to detect links with issues. This audit results in automated tickets to repair faulty links, followed by a repeat of the audit.

During DCN expansion, we must drain some fraction of the logical links. While Clos networks are resilient to some missing links, draining too many links simultaneously could result in dropping the network's available capacity below traffic demand. We therefore set a residual-capacity threshold, based on measured demand plus some headroom. We then divide an expansion into stages such that, during each stage, the residual capacity remains above this threshold. In our original approach, we partitioned the set of patch panels into C groups (as illustrated by different colors in Fig. 1), and only rewired links in one group of patch panels per stage. Then, at each stage, we would still have approximately $1-1/C$ of the pre-expansion capacity available.

Note that the expansion pipeline migrates the network, in stages, from an existing (old) topology to some new logical topology that connects the new blocks to the existing ones, subject to a set of constraints on the logical connectivity (§5.2.1 formalizes those constraints). However, there are many ways to construct a logical topology that meets our constraints, and our original, simple solution to these constraints typically required a lot of rewiring for an expansion. This, in turn, forced us to divide expansions into C stages, to ensure that we preserved at least $1-1/C$ of the pre-expansion capacity.

As a result, these expansions took a long time, especially if the pre-expansion network was highly utilized. In our experience, each stage takes considerable time, including draining, rewiring, cable auditing, BERTing, and undraining. If we were to expand a network with a requirement to preserve 90% residual capacity, at least 10 stages would be needed[5].

The root cause of the length of our original expansion pipeline is that it does not attempt to optimize the difference (in terms of wiring changes) between pre-expansion and post-expansion topologies. If we were able to minimize this difference, we could finish an expansion in fewer stages. This is related to the "incremental expansibility" property of a DCN topology. This property is easy to achieve for random-graph topologies such as Jellyfish [28]. However, for Clos networks, none of the existing topology solutions exhibits this property. This motivated us to look for a better approach, which we describe in the rest of this paper.

5 Minimal Rewiring for DCN Expansion

In this section, we describe a new "minimal rewiring" topology-design methodology for Clos networks, which not only achieves high network capacity and failure resiliency, but also minimizes rewiring for expansions. Our approach relies on Integer Linear Programming (ILP) to compute a logical topology. Our results compare well to existing Clos topology solutions [27, 5, 13, 33].

While we initially considered an ILP formulation at the granularity of switch and patch-panel ports, the resulting scale made solving infeasible. Instead, we use multiple phases, first solving a block-level ILP formulation (§5.2) and then performing a straightforward mapping onto a port-level solution (§5.4).

5.1 Definitions and notations

To rigorously formulate the minimal-rewiring topology-design problem, we introduce the following definitions and mathematical notations.

Table 1: Notations used in this paper

E_n, S_m, O_k	Server block n, spine block m, patch panel k
E_n^t	Middle block t in server block n
$G_k(E_n^t)$	Physical link-count for E_n^t via patch panel k
$G_k(S_m)$	Physical link-count for S_m via patch panel k
$b_k(E_n^t, S_m)$	Reference logical-topology link-count between E_n^t and S_m via patch panel O_k
$d_k(E_n^t, S_m)$	Desired logical-topology link-count between E_n^t and S_m via patch panel O_k
$p_{n,m}$	Mean number of links between a server block and a spine block
$q_{n,m}^t$	Mean number of links between a middle block and a spine block
n_g	Server block group index; or a set containing all the server block indices in the n_g-th group
m_g	Spine block group index; or a set containing all the spine block indices in the m_g-th group
k_g	Patch panel group index; or a set containing all the patch panel indices in the k_g-th group
E_{n_g}, S_{m_g}, O_{k_g}	Server block group n_g, spine block group m_g, patch panel group k_g
$E_{n_g}^t$	All the t-th middle blocks in E_{n_g}
$b_{k_g}(E_{n_g}^t, S_{m_g})$	Reference logical-topology link-count between $E_{n_g}^t$ and S_{m_g} via patch panel group O_{k_g}
$d_{k_g}(E_{n_g}^t, S_{m_g})$	Desired logical-topology link-count between $E_{n_g}^t$ and S_{m_g} via patch panel group O_{k_g}
x^+	$\max\{0, x\}$

Server Block: We use a server block as a unit of deploying network-switching capacity. On the order of 1000 servers can be connected to a server block via ToR switches. To avoid a single point of failure, we further divide a server block into independent *middle blocks*, typically four [6] (see Fig. 1); each middle block is controlled by a different routing controller. The uplinks of each ToR switch are evenly spread among the middle blocks in its server block. Even if one middle block is down, servers in the server block are still accessible via its other middle blocks. We assume that a DCN has N server blocks, each of which is represented by $E_n, n = 1, 2, ..., N$. We denote the middle blocks within server block E_n by $E_n^t, t = 1, 2, 3, 4$.

Spine Block: Spine blocks forward traffic among different server blocks. We use $S_m, m = 1, 2, ..., M$ to represent a spine block, where M is the total number of spine blocks.

Physical Topology: Assume there are K patch panels, each of which is represented by $O_k, k = 1, 2, ..., K$. We use $G_k(E_n^t)$ to represent the total number of physical links between the middle block E_n^t and the patch panel O_k. Then, the physical topology of server block E_n can be characterized by $\{G_k(E_n^t), k = 1, ..., K, t = 1, 2, 3, 4\}$[7].

Similarly, we use $G_k(S_m)$ to represent the total number of physical links between spine block S_m and patch panel O_k. Then, the physical topology of spine block S_m can be characterized by $\{G_k(S_m), k = 1, ..., K\}$.

Note that our networks are heterogeneous: different server blocks and spine blocks could have different physical topologies (see Table 2 for details).

Reference Topology: Our goal is to minimize rewiring with respect to the old logical topology, called the *reference topology*. We let $b_k(E_n^t, S_m)$ be the total number of reference-topology links between middle block E_n^t and spine block S_m

[5] In fact, since failed links and inexact load balance can cause topology imperfections, more than 10 stages would be required to ensure 90% residual capacity.

[6] Our results hold for any number of middle blocks.

[7] Here we use the term "topology" somewhat loosely to describe the cardinality of the connectivity between a block and a set of patch panels, rather than to describe either the detailed inter-block topology, or the internal topology within a block composed of multiple commodity switches.

that connect via patch panel O_k. Since the new server and spine blocks do not exist in the reference topology, for these blocks we simply set $b_k(E_n^t, S_m) = 0$.

Logical Topology: Our objective is to compute a new logical topology for the given physical topology. We use $d_k(E_n^t, S_m)$ to represent the total number of logical links between middle block E_n^t and spine block S_m that connect via patch panel O_k. As we will show shortly, as long as $\{d_k(E_n^t, S_m)\}_{k,m,n,t}$ satisfies a set of physical topology constraints, a polynomial-time algorithm can be used to map $\{d_k(E_n^t, S_m)\}_{k,m,n,t}$ to point-to-point configurations in the patch panels. Hence, the objective of the block-level ILP formulation is to compute $\{d_k(E_n^t, S_m)\}_{k,m,n,t}$.

5.2 ILP Formulation of Minimal Rewiring

Our ILP formulation consists of a set of constraints (§5.2.1) and an objective (§5.2.2).

5.2.1 Constraints for Logical Topology

Recall that our objective is to compute $\{d_k(E_n^t, S_m)\}_{k,m,n,t}$. We must impose a set of constraints on the solution, not only to ensure the compatibility of the logical topology with the underlying physical topology, but also to guarantee both high throughput and high failure resiliency.

Physical Topology Constraints: Recall that $d_k(E_n^t, S_m)$ is the total number of logical links between middle block E_n^t and spine block S_m connected via patch panel O_k. Clearly, it must be no larger than the total number of physical links $G_k(E_n^t)$ between O_k and E_n^t, and the total number of physical links $G_k(S_m)$ between O_k and S_m, i.e.,

$$0 \le d_k(E_n^t, S_m) \le \min\{G_k(E_n^t), G_k(S_m)\}. \tag{1}$$

To ensure high uplink bandwidth for all the server blocks, we require that each "populated" server block port[8] must connect to a spine block port. This is guaranteed by

$$\sum_{m=1}^{M} d_k(E_n^t, S_m) = G_k(E_n^t). \tag{2}$$

Note that constraint (2) requires that the total number of spine block ports must be no smaller than the total number of server block ports on each patch panel. Hence, it is possible that a physically-connected spine block port might not connect to any server block port, which can be expressed as:

$$\sum_{n=1}^{N} \sum_{t=1}^{4} d_k(E_n^t, S_m) \le G_k(S_m). \tag{3}$$

Capacity Constraints: In order to achieve high DCN capacity, which depends on load balance, we require the uplinks of each server block to be evenly distributed among all spine blocks. Specifically,

$$\lfloor p_{n,m} \rfloor \le \sum_{k=1}^{K} \sum_{t=1}^{4} d_k(E_n^t, S_m) \le \lceil p_{n,m} \rceil, \tag{4}$$

where $p_{n,m}$ is the mean number of links between a server block and a spine block. $p_{n,m} = |E_n||S_m|/(|S_1| + \cdots + |S_M|)$[9], where $|E_n|$ (or $|S_m|$) is the total number of ports in E_n (or S_m), $\lfloor p_{n,m} \rfloor$ is the largest integer that is no larger than $p_{n,m}$, and $\lceil p_{n,m} \rceil$ is the smallest integer that is no smaller than $p_{n,m}$.

Constraint (4) ensures high DCN capacity. In the ideal case where all $p_{n,m}$'s are integers, constraint (4) ensures that traffic between any two server blocks E_{n_1} and E_{n_2} can burst at full rate ($\min\{|E_{n_1}|, |E_{n_2}|\}$). Specifically, E_{n_1} and E_{n_2} can communicate at rate $\min\{p_{n_1,m}, p_{n_2,m}\}$ through the m-th spine block, and thus the total rate would be $\sum_{m=1}^{M} \min\{p_{n_1,m}, p_{n_2,m}\} = \min\{|E_{n_1}|, |E_{n_2}|\}$.

In the general case where some $p_{n,m}$'s are not integers, there must be some imbalance in the logical topology. Constraint (4) minimizes this imbalance.

Failure-Resiliency Constraints: While commodity switches are highly reliable, an entire middle block can fail as a unit due to software bugs in its routing controller. We also bring down an entire middle block occasionally to upgrade the switch stack softwares. In order for our DCN to be failure-resilient, we need to make sure that throughput remains as high as possible even under middle-block failures. This can be achieved by requiring the middle block links to be evenly distributed among the spine blocks. Specifically,

$$\lfloor q_{n,m}^t \rfloor \le \sum_{k=1}^{K} d_k(E_n^t, S_m) \le \lceil q_{n,m}^t \rceil, \tag{5}$$

where $q_{n,m}^t$ is the mean number of links between a middle block and a spine block[10]. $q_{n,m}^t = |E_n^t||S_m|/(|S_1| + \cdots + |S_M|) = p_{n,m}/4$ (assuming 4 middle blocks per server block).

Constraint (5) minimizes the capacity impact under middle block failures. In the ideal case where $q_{n,m}^t$'s are all integers, the throughput impact is exactly 25%. In the general case where some $q_{n,m}^t$'s are not integers, there must be some imbalance in the traffic between the middle blocks and the spine blocks. Constraint (5) minimizes this imbalance. Note that Constraint (5) cannot subsume (4), because all the decision variables are integers.

5.2.2 Minimal-Rewiring ILP Objective

In our ILP-based formulation, it is easy to add a minimal-rewiring objective function. Specifically, our block-level minimal-rewiring solver can be formulated as:

$$\begin{aligned} \min \quad & \sum_{k=1}^{K} \sum_{n=1}^{N} \sum_{t=1}^{4} \sum_{m=1}^{M} (b_k(E_n^t, S_m) - d_k(E_n^t, S_m))^+, \\ \text{subject to} \quad & (1) - (5), \end{aligned} \tag{6}$$

where $x^+ = \max\{0, x\}$. This objective function computes the total number of links to be rewired for changing the old (reference) topology to the new topology.

[8] Recall that, for cost reasons, we do not initially populate all the ports.

[9] We have assumed, when deriving $p_{n,m}$, that the number of spine block ports is no less than the number of server block ports. In fact, our formulation and the subsequent optimization techniques also apply to the cases where there are fewer spine block ports.

[10] Our formulas for p and q require trivial extensions to support heterogeneous link speeds; we omit these for reasons of space.

5.3 Benefits of Minimal Rewiring

We demonstrate the benefit of minimal rewiring using a simple example, comparing our minimal-rewiring approach against the *rotation striping* approach described in [33]. Rotation striping can be used to design Clos topologies for *homogeneous* DCNs of arbitrary size,whereas minimal-rewiring works with various block sizes.

Consider a DCN configuration with N server blocks and M spine blocks. Assume that each server block has X ports, and that there is only one patch panel[11]. Then, rotation striping can be expressed as Algorithm 1.

```
Input: DCN configuration parameters $N$, $M$, $X$
Output: A DCN topology
Label all the server block ports with different indices from
    $1,2,...,NX$. Note that the set of ports
    $\{(n-1)X+1,(n-1)X+2,...,nX\}$ corresponds to the
    $n$-th server block.
Connect port $e \in \{1,2,...,NX\}$ to the $\lceil e/M \rceil$-th port of the
    $((e-1)\%M+1)$-th spine block.
```

Algorithm 1: Rotation striping algorithm from [33]

We can quantify the *rewiring ratio* for a solution as the fraction of wires between server blocks and spine blocks in the pre-existing topology that must be disconnected during an expansion procedure.

Consider an expansion in which we add 1 server block and 1 spine block. It is easy to check that with rotation striping, only the first M server-block ports connect to their original peers, and thus the rewiring ratio would be $(NX-M)/NX$.

On the other hand, using minimal-rewiring[12], we can show that only $XN/(M+1)$ links need to be rewired, which corresponds to a rewiring ratio of $1/(M+1)$. This means that even if our expansion is executed in just a single stage, the capacity reduction is just $1/(M+1)$.

To the best of our knowledge, with the exception of Condor [25], none of the prior literature has incorporated patch-panel layer constraints, and Condor's constraint-satisfaction approach was unable to find solutions in most cases. In practice, we usually need more than one patch panels in order to connect all the server blocks to all the spine blocks, since the number of ports on each patch panel is limited. If one ignores the patch-panel layer, then the resulting DCN topology will be very likely not compatible with the underlying physical topology.

Consider rotation striping in an example with two patch panels (see Fig. 5). Each server block has six links, with three links connecting to the first patch panel and the other three connecting to the second one. There are four spine blocks, with two of them connecting to the first patch panel and the other two connecting to the second one. If we apply rotation striping here, there should be four logical links between the first server block and the first two spine blocks. Note that these four links can only be created through the first patch panel, because the first two spine blocks only connect to the first patch panel. However, this is impossible, as there are only three physical links between the first server block and the first patch panel.

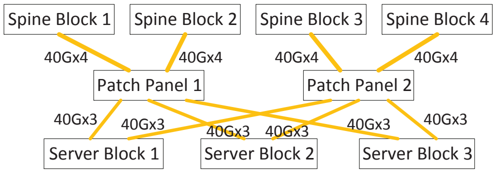


Figure 5: A counterexample for which rotation striping fails.

Our approach incorporates the patch-panel layer via three physical constraints (1)-(3). These three constraints ensure that any solution $\{d^*_k(E^t_n, S_m)\}_{k,m,n,t}$ of (6) can be mapped to port-to-port configurations in the patch panels as we describe below. (Note that (6) may yield multiple solutions.)

5.4 Creating port-to-port mappings

Our ILP formulation tells us the block-level link count between each middle block E^t_n and each spine block S_m in each patch panel O_k, as in (6), but not how individual ports must be connected. We thus developed a straightforward algorithm to compute these port-to-port mappings.

The algorithm's input consists of the block-level link counts $b_k(E^t_n, S_m)$ and $d^*_k(E^t_n, S_m)$ for the pre-expansion and post-expansion topologies, respectively. We use two passes, both of which iterate over all pairs of middle blocks and spine blocks:

Pass 1: disconnect links as necessary: For each patch panel O_k, note that an expansion changes the link count between middle block E^t_n and spine block S_m from $b_k(E^t_n, S_m)$ to $d^*_k(E^t_n, S_m)$. Therefore, if $b_k(E^t_n, S_m) \leq d^*_k(E^t_n, S_m)$, we simply preserve all pre-existing links; if $b_k(E^t_n, S_m) > d^*_k(E^t_n, S_m)$, we can disconnect any $(b_k(E^t_n, S_m)$ $-d^*_k(E^t_n, S_m))$ of the pre-existing links. This pass disconnects $\sum_{n=1}^{N}\sum_{t=1}^{4}\sum_{m=1}^{M}(b_k(E^t_n, S_m) - d^*_k(E^t_n, S_m))^+$ links.

Pass 2: connect new links: After the first pass, $\min\{b_k(E^t_n, S_m), d^*_k(E^t_n, S_m)\}$ links remain between E^t_n and S_m. For any block pair E^t_n and S_m with less than $d^*_k(E^t_n, S_m)$ links, we can arbitrarily pick $d^*_k(E^t_n, S_m) - b_k(E^t_n, S_m)$ non-connected ports from E^t_n and S_m respectively, and interconnect them. Feasibility is guaranteed by the physical topology constraints (1)-(3).

5.5 Challenge: Solver Scalability

We use integer linear programming to formulate our minimal-rewiring solver, because our topology-design problem is NP-Complete. Specifically, the decision variables $d_k(E^t_n, S_m)$ contain three dimensions (middle-block dimension E^t_n, spine-block dimension S_m, and patch-panel di-

[11] Rotation striping does not consider the patch-panel layer, which is equivalent to setting the number of patch panels to be 1.

[12] Rotation striping can only guarantee the constraints (1)-(4). Hence, *for this example*, we also only impose those constraints on our minimal rewiring solver.

mension k), and the constraints (2)(3)(5) are essentially for the 2-marginal sums $\sum_k d_k(E_n^t, S_m)$, $\sum_{E_n^t} d_k(E_n^t, S_m)$ and $\sum_{S_m} d_k(E_n^t, S_m)$. In the literature, this is called the Three-Dimensional Contingency Table (3DCT) problem, and has been proven to be NP-Complete [20]. Having failed to find a polynomial-time algorithm for our problem, we decided to use ILP, as there are many readily-available commercial ILP solvers, e.g., Gurobi [17], Google Optimization Tools [12].

However, our problem size is so large that none of the existing commercial solvers scales well. For example, one DCN configuration we evaluate (see §9.7.1) contains 77 server blocks of three kinds, 68 spine blocks of two kinds, and 256 patch panels. Without any optimization, this leads to about 400000 decision variables, and the ILP solver ran for a day without generating a solution. As shown in §9.3, for the 4500 trials we ran without optimization, we could only solve 32% within a 3-hour deadline. (Longer timeouts yield little improvement.)

6 Block Aggregation

To improve the scalability of our minimal-rewiring solver, we developed a block aggregation technique. Block aggregation significantly reduces the total number of decision variables in (6), and thus greatly improves solver scalability.

6.1 ILP Formulation with Block Aggregation

The idea behind block aggregation is to group patch panels, server blocks, spine blocks, and then aggregate decision variables within each group.

Patch-Panel Group: Two patch panels k_1, k_2 belong to the same group if and only if they have the same number of physical links to each middle block and each spine block, i.e., $G_{k_1}(E_n^t) = G_{k_2}(E_n^t), G_{k_1}(S_m) = G_{k_2}(S_m)$ for any n, t, m.

Server-Block Group: Two server blocks n_1, n_2 belong to the same group if and only if they have the same physical topology, i.e., $G_k(E_{n_1}^t) = G_k(E_{n_2}^t)$ for any k, t.

Spine-Block Group: Two spine blocks m_1, m_2 belong to the same group if and only if they have the same physical topology, i.e., $G_k(S_{m_1}) = G_k(S_{m_2})$ for any k.

Assume that these definitions yield K_g patch-panel groups, N_g server-block groups, and M_g spine-block groups. Then, we can define an aggregated decision variable $d_{k_g}(E_{n_g}^t, S_{m_g})$ for the k_g-th patch-panel group, the n_g-th server block group, and the m_g-th spine block group as follows:

$$d_{k_g}(E_{n_g}^t, S_{m_g}) = \sum_{k \in k_g} \sum_{n \in n_g} \sum_{m \in m_g} d_k(E_n^t, S_m). \tag{7}$$

Here, we have abused the notation, and used k_g, n_g, and m_g to represent both the indices and the actual groups.

With the aggregated decision variables $d_{k_g}(E_{n_g}^t, S_{m_g})$, the original constraints (1)-(5) also need to be aggregated:

$$0 \leq d_{k_g}(E_{n_g}^t, S_{m_g}) \leq |k_g||n_g||m_g| \min\{G_k(E_n^t), G_k(S_m)\}, \tag{8}$$

$$\sum_{m_g=1}^{M_g} d_{k_g}(E_{n_g}^t, S_{m_g}) = |k_g||n_g| G_k(E_n^t), \tag{9}$$

$$\sum_{n_g=1}^{N_g} \sum_{t=1}^{4} d_{k_g}(E_{n_g}^t, S_{m_g}) \leq |k_g||m_g| G_k(S_m), \tag{10}$$

$$|n_g||m_g| \lfloor p_{n,m} \rfloor \leq \sum_{k_g=1}^{K_g} \sum_{t=1}^{4} d_{k_g}(E_{n_g}^t, S_{m_g}) \leq |n_g||m_g| \lceil p_{n,m} \rceil, \tag{11}$$

$$|n_g||m_g| \lfloor q_{n,m}^t \rfloor \leq \sum_{k_g=1}^{K_g} d_{k_g}(E_{n_g}^t, S_{m_g}) \leq |n_g||m_g| \lceil q_{n,m}^t \rceil. \tag{12}$$

In the aggregated constraints, k, n, m are arbitrary patch panel, server block, and spine block indices, drawn from the k_g-th patch-panel group, the n_g-th server-block group, and the M_g-th spine-block group, respectively. In fact, the values $G_k(E_n^t)$, $G_k(S_m), p_{n,m}, q_{n,m}^t$ are all the same, as long as k, n, m are chosen in the same group, respectively. Here, we also view k_g, n_g, and m_g as groups, and have used $|k_g|, |n_g|, |m_g|$ to represent the sizes of these groups.

With block aggregation, we can thus rewrite the optimization problem (6) as follows:

$$\begin{aligned} \min \quad & \sum_{k_g=1}^{K_g} \sum_{n_g=1}^{N_g} \sum_{t=1}^{4} \sum_{m_g=1}^{M_g} (b_{k_g}(E_{n_g}^t, S_{m_g}) - d_{k_g}(E_{n_g}^t, S_{m_g}))^+, \\ \text{subject to} \quad & (8) - (12), \end{aligned} \tag{13}$$

where $b_{k_g}(E_{n_g}^t, S_{m_g}) = \sum_{k \in k_g} \sum_{n \in n_g} \sum_{m \in m_g} b_k(E_n^t, S_m)$.

Compared to (6), the total number of decision variables in (13) is significantly reduced, from $\Theta(NKM)$ to $\Theta(N_g K_g M_g)$. Thus, the complexity of solving (13) is significantly lower than that of (6).

6.2 Variable Deaggregation

After obtaining a solution $d_{k_g}^*(E_{n_g}^t, S_{m_g})$ for (13), we still need to decompose the solution to $d_k^*(E_n^t, S_m)$. Specifically, we need to solve the following problem.

$$\begin{aligned} \min \quad & \sum_{k=1}^{K} \sum_{n=1}^{N} \sum_{t=1}^{4} \sum_{m=1}^{M} (b_k(E_n^t, S_m) - d_k(E_n^t, S_m))^+, \\ \text{subject to} \quad & \sum_{k \in k_g} \sum_{n \in n_g} \sum_{m \in m_g} d_k(E_n^t, S_m) = d_{k_g}^*(E_{n_g}^t, S_{m_g}) \\ & \text{and } (1) - (5). \end{aligned} \tag{14}$$

If there were no constraints (1)-(5) in (14), we could easily compute a solution $d_k^*(E_n^t, S_m)$ in polynomial time using an algorithm similar to the one in §5.4. Because of these constraints, solving (14) becomes much more challenging. In fact, it is not trivial to prove that (14) always has a solution. In the literature, (14) is closely related to the Integer-Decomposition property [2]. In general, the Integer-Decomposition property does not always hold (an example is provided in [32]). Fortunately, thanks to our problem structure, we are able to rigorously prove that (14) indeed has a solution. Specifically, we build an integer decomposition theory specifically for our problem, and prove that a decomposition satisfying (1)-(5) can be found iteratively in polynomial time. The details are available in [32].

Given that (14) has solutions, the next question is to find one that minimizes the objective function. The simplest approach is to directly solve it using integer programming. However, this will destroy the complexity-reduction of block aggregation, as (14) is of exactly the same size as (6).

Our approach is to decompose (14) into $K_g + N_g + M_g$ smaller ILP problems. Specifically, we can decompose patch-panel groups, spine-block groups, and server-block groups by solving three ILPs in separate steps. In each step, different block groups can be completely decoupled, and thus the three ILPs can be further decomposed into K_g, N_g, and M_g ILPs, respectively. These smaller ILP problems are much easier to solve compared to (14). However, as shown in §9.3, computing these smaller ILP problems sequentially can still be slow. To further improve scalability, we map these smaller problems to polynomial-complexity min-cost-flow problems. This min-cost-flow-based decomposition can guarantee a solution that satisfies all constraints, but not rewiring optimality. For details, please refer to [32] .

6.3 Impact on Scalability & Optimality

Theoretically, and as confirmed in §9.4, the total number of rewires would be higher with block aggregation. We are essentially breaking one optimization problem (6) into two problems (13) and (14). Solving (13) and (14) will generate a solution satisfying all constraints in (6), but which might not be optimal wrt. the minimal-rewiring objective (6). Whenever the solver without block aggregation succeeds, it always achieve the smallest (best) rewiring ratio. However, scalability without block aggregation is poor. For a total of 4500 synthesized DCN expansion configurations, only 32% can be solved within a 3-hour limit. (Longer timeouts yield little improvement.)

Breaking (6) into smaller ILPs does not completely solve the solver scalability issue. We may still encounter some intractable ILPs while solving (14) (see §9.3). Our polynomial-time min-cost-flow based variable-deaggregation algorithm solves the scalability issue. However, it may also introduce some sub-optimality in the rewiring ratio (see §9.4 for the detailed comparison). Thus, we face a tradeoff between solver scalability and rewiring optimality, which we address in the next section.

7 Parallel Solving Architecture

While block aggregation makes it feasible to solve most DCN expansion configurations, it creates a tradeoff between solver scalability and rewiring optimality, depending on how one chooses a *strategy* for block aggregation. Block-aggregation strategies define choices for each of several aggregation layers (patch-panel, server-block, or spine-block), and for the decomposition technology (ILP or a min-cost-flow approximate algorithm) applied at each layer.

How can we choose the best strategy among all the options, given that the tradeoff between optimality and solver run-time is unknown when we start a solver? We observe that since we care more about finding a solution within an elapsed time limit, and less about the total computational resources we use, our best approach is to run, in parallel, a solver instance for each of the options, and then choose the best result, based on a scoring function, among the solvers that complete within a set deadline. We can define scores based simply on rewiring ratio, or on residual bandwidth during expansion, or some combination. Fig. 6 shows the parallel-solver architecture.

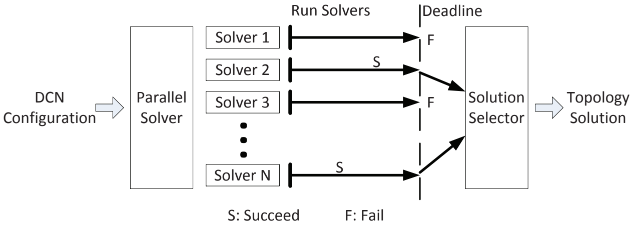


Figure 6: Software architecture of the parallel solver.

8 Changes to the Expansion Pipeline

The introduction of minimal rewiring requires several changes to the DCN expansion pipeline shown in Fig. 4.

Without minimal rewiring, it is fairly easy for an experienced network engineer to determine the number of stages required for an expansion; because almost all logical links need to be rewired, we must use C stages to maintain a residual capacity of $1 - 1/C$. With minimal rewiring, however, one cannot know the number of rewired links before running the solver; based on results in §9.4, the fraction could range from 0 to 30%.

Therefore, we add an automated **expansion planner** step to the front of the pipeline in Fig. 4. The planner first uses the minimal-rewiring solver to generate a post-expansion topology, then iteratively finds the lowest C, starting at $C = 1$, which preserves the desired residual capacity threshold during the expansion (recall from §4 that this threshold is a function of forecasted demand, existing failures, and some headroom). For each C, the planner divides the patch panels into C groups, tentatively generates the C intermediate topologies that would result from draining the affected links within a group, and evaluates those topologies against the threshold. If capacity cannot be preserved for all intermediate topologies, the planner increments C and tries again. Once a suitable C is found, the rest of the pipeline is safe to execute.

Without minimal rewiring, we simply drain all links for the patch panels covered by a stage. Minimal rewiring allows us to preserve more capacity because we only have to drain a subset of the links, rather than an entire patch panel. However, an operator can accidentally rewire the wrong link, or dislodge one inadvertently, a problem that does not occur when we drain entire panels. Therefore, we built a link-status monitor that alerts the operator if an active (undrained) link is disconnected. Since Clos networks by design tolerate some link failures, this allows the operator enough time to repair the problem without affecting users.

9 Experimental Results

We have been successfully using this minimal-rewiring approach for our own DCNs since early 2017[13]. In order to demonstrate its benefits over a wide range of scales, we evaluated our approach using a set of synthetic DCN topologies (including some similar to our real DCNs), and show the effect of block aggregation on solver run-time and rewiring ratio. We also show that our approach preserves most of the capacity of the original network during expansions, based on several workloads and under some failure scenarios.

9.1 Synthesizing DCN Configurations

To evaluate our approach over a wide range of initial topologies, we synthesized thousands of configurations that were consistent with our past deployment experience. In particular, since we must support heterogeneity among server blocks and spine blocks, we synthesized configurations using three types of server blocks and two types of spine block, in various combinations. Every configuration includes exactly one border block, analogous to a server block but used for external connectivity.

Our synthesized DCN configurations always have 256 patch panels, located in 4 sets, each with 64 patch panels. We use patch panels with 128 server-block-facing ports and 128 spine-block-facing ports, so the entire patch-panel layer supports up to 64 512-port server blocks.

Table 2 lists the physical-topology parameters for the different block types. For example, a Type-1 server block has 512 up links, evenly distributed among all 256 patch panels. Type-2 & Type-3 server blocks are "light" versions of the Type-1 server block, with fewer uplinks; they can be upgraded to Type-1 server blocks. Note that a Type-3 server block only connects to 128 patch panels. We divide the 256 patch panels into two partitions, and connect the Type-3 server blocks to the two partitions by rotation. We also connect the Type-1 & Type-2 spine blocks by rotation.

Table 2: Server Block and Spine Block Types.

Block Type	Uplinks	Connected Patch Panels
Type-1 Server Block	512	64 Per Set×4 Sets
Type-2 Server Block	256	64 Per Set×4 Sets
Type-3 Server Block	256	32 Per Set×4 Sets
Border Block	1024	64 Per Set×4 Sets
Type-1 Spine Block	128	64 Per Set×1 Set
Type-2 Spine Block	512	64 Per Set×1 Set

We generate a total of 2250 initial configurations (pre-expansion "reference topologies" as defined in §5.1) from all possible combinations of $\{3,6,...,30\}$ Type-1 server blocks, $\{2,4,...,20\}$ Type-2 server blocks, $\{3,6,...,30\}$ Type-3 server blocks, and $\{8,16\}$ Type-1 spine blocks, with the remainder of the necessary spine-block ports as Type-2 spine blocks (the total number of server-block and spine-block ports must match). We omit any configuration that would require more patch-panel ports than we have available.

These pre-expansion topologies can be generated using our minimal rewiring solver, by simply ignoring the objective function. We run all the configurations in parallel, and allocate 4 CPUs and 16G RAM for each configuration. With block aggregation enabled, all 2250 topologies can be computed within 10 seconds.

9.2 Expansion Benchmarks

As shown in Fig. 3, we support two types of DCN expansions. We construct two benchmarks for each of our 2250 reference topologies:

Benchmark Suite 1: We upgrade two Type-2 server blocks in the reference topology to Type-1 server blocks.

Benchmark Suite 2: We expand the reference topology by one Type-3 server block.

We end up with $2250 \times 2 = 4500$ total target topologies. For each of these, we ran the minimal-rewiring solver with three aggregation strategies:

1. No aggregation.
2. Block aggregation, decomposing the spine-block and server-block layers using ILP, while decomposing the patch-panel layer using MIN_COST_FLOW (See technical report [32] for details) .
3. Block aggregation, decomposing all three layers using MIN_COST_FLOW.

9.3 Solver scalability

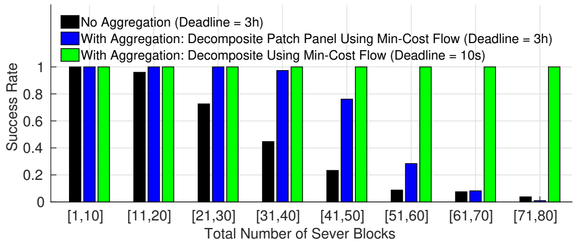


Figure 7: Minimal-rewiring solver success rate.

Fig. 7 plots success rate for our minimal-rewiring solver with different aggregation strategies, grouped by the total number of server blocks; the bars show the fraction of topologies solved for each group. With the third aggregation strategy, we can solve *all* test cases within 10 seconds; this strategy only needs to solve one ILP for the aggregated problem, while using polynomial algorithms for all decompositions. The first strategy scales poorly, and can only solve 32% of the test cases even if we increase the deadline to 3 hours; even for small DCNs (11–20 server blocks), it sometimes times out. The second strategy can solve 67% of the test cases, but we start seeing timeouts for DCNs with 31–

[13] Even though we have a formal proof that deaggregation always works, it does not guarantee optimality; our experience shows that our approach does work in practice.

40 server blocks. For all strategies, setting timeouts above 3 hours yields little improvement.

The differences between the second and third strategies shows that variable deaggregation using ILP could take significant run time. (If we also use ILP to deaggregate the patch-panel layer with the second strategy, only 3% of the test cases can be solved.) However, deaggregation via ILP is still useful, because as we discuss next, it can generate more-optimal rewiring solutions.

9.4 Rewiring Ratio

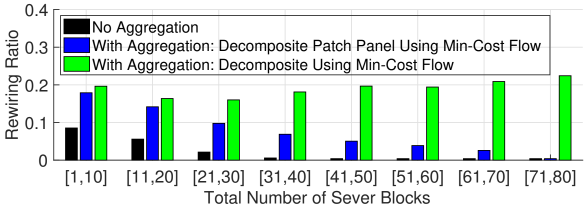


Figure 8: Mean rewiring ratio vs. aggregation strategy.

For the same set of test cases, Fig. 8 plots the rewiring ratios, again grouped by the number of server blocks; here, the bars show the mean value for each group. The third aggregation strategy, while it has the best run time, also leads to the highest (worst) rewiring ratio, often close to 20%; this motivates our use of ILP for decomposition whenever we can tolerate the run time.

9.5 Effectiveness of the parallel solver

§7 described how we use a parallel solver to strike a balance between scalability and rewiring optimality. For each of our benchmarks, we ran, in parallel, solvers with the three different aggregation strategies, with a 3-hour timeout. Because the third strategy works quickly for all instances, this parallel approach always succeeds. It also achieves the best rewiring ratio available within a 3-hour budget. Fig. 9 plots the CDF of the parallel solvers rewiring ratio. For about 82% of the DCN configurations in the first benchmark suite, and about 93% in the second suite, we get solutions with ratios under 20%. (The first suite tends to yield a higher ratio, because the total number of newly added server-block physical links in the first suite is twice that in the second suite.)

Note that the rewiring ratio for our prior approach was always 1.0 – we always replaced all patch-panel jumpers. Fig. 9 shows that minimal rewiring saves us a lot of cost and time; the median rewiring ratio is about 22× better for the first suite and about 38× better for the second suite.

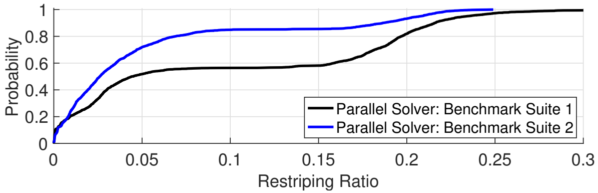


Figure 9: CDFs of rewiring ratios for the parallel solver.

9.6 Topology Capacity Analysis

In addition to solver scalability and rewiring optimality, we also must preserve sufficient capacity of the intermediate topologies during expansion. This requires us to quantify "capacity." While our DCNs experience a variety of traffic patterns, we specifically evaluate the maximum achievable capacity under one-to-all and one-to-one traffic, which we rigorously define as:

One-to-all capacity: $T_{\textbf{1-all}}$. Assume that a source server block E_n is sending traffic to all other server blocks, with its demand proportional to the ingress bandwidth of the destination blocks. We increase the traffic until some DCN links are fully utilized. Let $T^n_{\text{1-all}}$ be the ratio between the egress traffic under these assumptions and the best-case egress DCN bandwidth of the server block E_n. $T^n_{\text{1-all}}$ characterizes the one-to-all DCN capacity for server block E_n. We can then define $T_{\text{1-all}} = \min_{n=1}^{N} T^n_{\text{1-all}}$ as the one-to-all capacity of the entire DCN.

One-to-one capacity: $T_{\textbf{1-1}}$. Assume that a server block E_{n_1} is sending traffic to another server block E_{n_2}. We increase the traffic until some DCN links are fully utilized. Let $T^{n_1,n_2}_{\text{1-1}}$ be the ratio between the traffic sent and the minimum, over the two server blocks E_{n_1} and E_{n_2}, of their ingress capacity. $T^{n_1,n_2}_{\text{1-1}}$ characterizes the one-to-one DCN capacity for the server blocks E_{n_1} and E_{n_2}. We can then define $T_{\text{1-1}} = \min_{n_1 \neq n_2} T^{n_1,n_2}_{\text{1-1}}$ as the one-to-one capacity of the entire DCN.

We evaluate $T_{\text{1-all}}$ and $T_{\text{1-1}}$ under two different scenarios:

Steady-state no-failures capacity: Recall that we imposed constraint (4) when computing a topology, which ensures high capacity in the non-expansion steady state, without any failures. Ideally, if the $p_{n,m}$'s in (4) were all integers, we would have $T_{\text{1-all}} = 1$ and $T_{\text{1-1}} = 1$. (Neither $T_{\text{1-all}}$ or $T_{\text{1-1}}$ can be larger than 1). Fig. 10 plots CDFs of $T_{\text{1-all}}$ and $T_{\text{1-1}}$ based on the 4500 post-expansion topologies (benchmark suite 1 + benchmark suite 2). Note that both $T_{\text{1-all}}$ and $T_{\text{1-1}}$ are fairly close to 1.

Steady-state capacity under middle block failure: Recall that we imposed constraint (5), to ensure the highest possible capacity if one middle block fails. Ideally, for our typical case of 4 middle blocks, if the $q^t_{n,m}$'s in (5) were all integers, we would have $T_{\text{1-all}} = 0.75$ and $T_{\text{1-1}} = 0.75$, no matter which middle block fails. (Given one failure, neither $T_{\text{1-all}}$ or $T_{\text{1-1}}$ can be larger than 0.75. Fig. 10 also plots the CDFs of $T_{\text{1-all}}$ and $T_{\text{1-1}}$ under middle block failures. Note that both $T_{\text{1-all}}$ and $T_{\text{1-1}}$ are fairly close to 0.75.

9.6.1 Residual Capacity during Expansion

During an expansion, we must disconnect some links, which reduces DCN capacity. We are interested in the residual capacity of the DCN in this state which clearly depends on the rewiring ratio. Fig. 11 shows scatter plots of $T_{\text{1-all}}$ and $T_{\text{1-1}}$ for each (rewiring ratio, residual capacity) pair, based on the two benchmark suites. This figure assumes we do all expan-

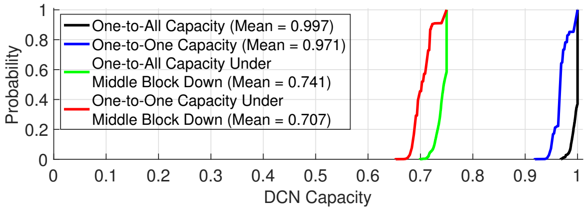
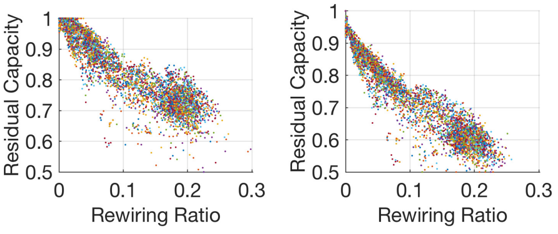


Figure 10: CDFs of One-to-All and One-to-One capacity.

sion in a single stage, even if the residual capacity is lower than we can accept in reality.

The residual capacity decreases approximately linearly with the rewiring ratio. Note that the residual capacity decreases faster than the rewiring ratio increase, because, the rewired DCN links might not be evenly distributed among different server blocks. As a result, some server blocks could suffer more capacity reduction than others.



(a) $T_{\text{1-all}}$. (b) $T_{\text{1-1}}$.

Figure 11: Rewiring ratio vs. *residual capacity assuming a 1-stage expansion.*

9.7 Number of expansion stages

In practice, we might not be able to do an expansion in just one stage while preserving sufficient residual capacity. §8 described an expansion planner that determines the number of stages required. We evaluated the number of stages saved by using minimal rewiring.

Prior to minimal rewiring, we typically did 4-stage expansions. If the topologies were perfect, with 4 stages we could preserve a residual one-to-all capacity of 0.75, but in practice we cannot achieve perfect balance; we have found it feasible and sufficient to preserve a residual capacity of 0.7.

Table 3 shows how many stages are needed to preserve a residual one-to-all capacity of 0.7, with minimal rewiring, for each of the 4500 benchmarks, based on solver strategy. With strategies 1 and 2, many test cases require four stages, because the solvers time out before finding 1- or 2-stage expansions. With strategy 3, almost all cases require at most 2 stages. The parallel solver finds 1- or 2-stage expansions for all test cases (because the few cases for which strategy 3 requires four stages are handled better by the other strategies), and usually does better than strategy 3 (because if the other strategies succeed within the deadline, they yield better rewiring ratios.) Overall, the parallel solver needs an average of 1.29 stages, vs. 4 stages for our prior approach.

Table 3: Number of expansion stages required.

Number of expansion stages:	1	2	4
Aggregation Strategy (1)	1598	34	2868
Aggregation Strategy (2)	2668	416	1416
Aggregation Strategy (3)	1582	2914	4
Parallel Solver	3176	1324	0

Cells show # of test cases that need given # of stages.

9.7.1 Concrete example

A concrete (arbitrary, but realistic) example demonstrates the benefits of minimal rewiring. Assume a pre-expansion DCN with 30 Type-1, 20 Type-2, 27 Type-3 server blocks, 1 border block, and 16 Type-1, 52 Type-2 spine blocks, which we expand by one Type-3 server block. Without minimal rewiring, we must rewire all the 28056 logical links, in four stages. With minimal rewiring, we need to rewire only 6063 of 28056 links (ratio $= 0.216$), in two stages. Due to the scale of this example, only the third aggregation strategy succeeds. Table. 4 shows how this maintains mid-expansion capacity (in italics) higher than our prior approach (in bold), and completes in two stages rather than four.

Table 4: Example: minimal rewiring vs. prior approach.

	Stage during expansion timeline					
	Pre	1	2	3	4	Post
Prior approach	0.94	**0.70**	**0.70**	**0.70**	**0.70**	0.94
Min. Rewiring	0.94	*0.78*	*0.78*	0.94	0.94	0.94

Cells show one-to-all capacity $T_{\text{1-all}}$ during expansion.

10 Conclusion

We have demonstrated that it is, in fact, feasible to do fine-grained expansions of heterogeneous Clos DCNs, at large scale, while preserving substantial residual capacity during an expansion, and with a significant reduction in the amount of rewiring (compared to prior approaches). We described how we use a patch-panel layer to reduce the physical complexity of DCN expansions. We then described an ILP formulation that allows us to minimize the amount of rewiring, a block-aggregation approach that allows scaling to large networks, and a parallel solver approach that yields the best tradeoff between elapsed time and rewiring ratio. Our overall approach flexibly handles heterogenous switch blocks, and enforces "balance constraints that guarantee both high capacity and high failure resiliency. We evaluated our approach on a wide range of DCN configurations, and found on average that it allows us to do expansions in 1.29 stages, vs. 4 stages as previously required.

11 Acknowledgements

We thank Google colleagues for their insights and feedback, including Bob Felderman, David Wetherall, Keqiang He, Yadi Ma, Jad Hachem, Jayaram Mudigonda, and Parthasarathy Ranganathan. We also thank our shepherd Ankit Singla, and the NSDI reviewers.

References

[1] Al-Fares, M., Loukissas, A., and Vahdat, A. A Scalable, Commodity Data Center Network Architecture. In *ACM SIGCOMM* (August 2008).

[2] Baum, S., and Trotter, L. E. Integer rounding and polyhedral decomposition for totally unimodular systems. *Optimization and Operations Research 157* (October 1978), 15–23.

[3] Boyd, S., Ghosh, A., and Magnani, A. Branch and bound methods. *Notes for EE392o, Stanford University* (November 2003).

[4] Camarero, C., Martinez, C., and Beivide, R. Random Folded Clos Topologies for Datacenter Networks. In *IEEE International Symposium on High Performance Computer Architecture* (February 2017).

[5] Cisco. Cisco data center spine-and-leaf architecture: Design overview. *Cisco White Paper* (2016).

[6] Clos, C. A study of non-blocking switching networks. *Bell System Technical Journal 32* (March 1953), 406–424.

[7] Curtis, A. R., Carpenter, T., Elsheikh, M., Lopez-Ortiz, A., and Keshav, S. REWIRE: An Optimization-based Framework for Data Center Network Design. In *INFOCOM* (March 2012).

[8] Curtis, A. R., Keshav, S., and Lopez-Ortiz, A. LEGUP: Using Heterogeneity to Reduce the Cost of Data Center Network Upgrades. In *ACM CoNEXT* (November 2010).

[9] Dinitz, M., Schapira, M., and Valadarsky, A. Explicit Expanding Expanders. *Algorithmica 78* (August 2017), 1225–1245.

[10] Farrington, N., and Andreyev, A. Facebook's data center network architecture. *IEEE Optical Interconnects Conference* (May 2013).

[11] Farrington, N., Porter, G., Radhakrishnan, S., Bazzaz, H. H., Subramanya, V., Fainman, Y., Papen, G., and Vahdat, A. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *SIGCOMM* (August 2010).

[12] Google Inc. Google Optimization Tools. https://developers.google.com/optimization/.

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

[14] 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 *SIGCOMM* (August 2009).

[15] 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 *SIGCOMM* (August 2008).

[16] Guo, D., Chen, T., Li, D., Li, M., Liu, Y., and Chen, G. Expandable and cost-effective network structures for data centers using dual-port servers. *IEEE Transactions on Computers 62* (July 2013), 1303–1317.

[17] Gurobi. www.gurobi.com.

[18] Gyarmati, L., Gulyas, A., Sonkoly, B., Trinh, T. A., and Biczok, G. Free-scaling your data center. *Computer Networks 57* (June 2013), 1758–1773.

[19] Higbie, C. Point-to-Point Data Center Cable Will Cost You in the Long Run. *Network World* (Feb. 2012). https://www.networkworld.com/article/2186410/tech-primers/point-to-point-data-center-cable-will-cost-you-in-the-long-run.html.

[20] Irving, R. W., and Jerrum, M. R. Three-dimensional statistical data security problems. *SIAM Journal on Computing 23* (February 1994), 170–184.

[21] Li, D., Guo, C., Wu, H., Tan, K., Zhang, Y., and Lu, S. FiConn: Using Backup Port for Server Interconnection in Data Centers. In *IEEE INFOCOM* (April 2009).

[22] Li, Z., Guo, Z., and Yang, Y. Bccc: An expandable network for data centers. *IEEE/ACM Transactions on Networking 24* (December 2016), 3740–3755.

[23] Liu, V., Halperin, D., Krishnamurthy, A., and Anderson, T. F10: A Fault-Tolerant Engineered Network. In *NSDI* (April 2013).

[24] Rajasekaran, S. On the convergence time of simulated annealing. *Research Report MS-CIS-90-89, University of Pennsylvania, Department of Computer and Information Science* (November 1990).

[25] Schlinker, B., Mysore, R. N., Smith, S., Mogul, J. C., Vahdat, A., Yu, M., Katz-Bassett, E., and Rubin, M. Condor: Better Topologies Through Declarative Design. In *SIGCOMM* (August 2015).

[26] SIEMON. Data Center Cabling Considerations: Point-to-Point vs. Structured Cabling. http://www.siemon.com/us/white_papers/09-06-18-data-center-point-to-point-vs-structured-cabling.asp.

[27] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HLZLE, U., STUART, S., AND VAHDAT, A. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter. In *SIGCOMM* (September 2015).

[28] SINGLA, A., HONG, C.-Y., POPA, L., AND GODFREY, P. B. Jellyfish: Networking Data Centers, Randomly. In *NSDI* (April 2012).

[29] VALADARSKY, A., SHAHAF, G., DINITZ, M., AND SCHAPIRA, M. Xpander: Towards Optimal-Performance Datacenters. In *CoNEXT* (December 2016).

[30] WALRAED-SULLIVAN, M., VAHDAT, A., AND MARZULLO, K. Aspen Trees: Balancing Data Center Fault Tolerance, Scalability and Cost. In *CoNEXT* (December 2013).

[31] YU, Y., AND QIAN, C. Space shuffle: A scalable, flexible, and high-bandwidth data center network. *IEEE Transactions on Parallel and Distributed Systems 27* (February 2016), 3351–3365.

[32] ZHAO, S., WANG, R., ZHOU, J., ONG, J., MOGUL, J. C., AND VAHDAT, A. Minimal Rewiring: Efficient Live Expansion for Clos Data Center Networks: Extended Version. Online: https://ai.google/research/pubs/pub47492.

[33] ZHOU, J., TEWARI, M., ZHU, M., KABBANI, A., POUTIEVSKI, L., SINGH, A., AND VAHDAT, A. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *EuroSys* (April 2014).