Condor: Better Topologies Through Declarative Design

Brandon Schlinker[1,2], Radhika Niranjan Mysore[1], Sean Smith[1], Jeffrey C. Mogul[1], Amin Vahdat[1], Minlan Yu[2], Ethan Katz-Bassett[2], and Michael Rubin[1]

[1]Google, Inc. — [2]University of Southern California

ABSTRACT

The design space for large, multipath datacenter networks is large and complex, and no one design fits all purposes. Network architects must trade off many criteria to design cost-effective, reliable, and maintainable networks, and typically cannot explore much of the design space. We present Condor, our approach to enabling a rapid, efficient design cycle. Condor allows architects to express their requirements as constraints via a Topology Description Language (TDL), rather than having to directly specify network structures. Condor then uses constraint-based synthesis to rapidly generate candidate topologies, which can be analyzed against multiple criteria. We show that TDL supports concise descriptions of topologies such as fat-trees, BCube, and DCell; that we can generate known and novel variants of fat-trees with simple changes to a TDL file; and that we can synthesize large topologies in tens of seconds. We also show that Condor supports the daunting task of designing multi-phase network expansions that can be carried out on live networks.

CCS Concepts

•Networks → Topology analysis and generation; Physical topologies; *Data center networks;*

Keywords

Topology design; expandable topologies; SLO compliance

1. INTRODUCTION

During the design of a datacenter topology, a network architect must balance operational goals with technical, physical and economic constraints. An ideal network would be resilient to any failure, fulfill every application's requirements, and remain under budget. Unfortunately, the underlying constraints are fundamentally at odds with one another. No single network topology design is optimal for all use cases [2], due (among other things) to differences in scale and workloads, and changes in the cost and performance of components (such as switch silicon and opto-electronics).

Hence, architects must understand and make tradeoffs between the cost and availability of technologies on the marketplace, the complexity of wiring, limits on physical failure domains, and the fundamental limitations of routing algorithms. Today, balancing these tradeoffs is more art than science. Architects continually go back to the drawing board to adapt and refine existing topologies, or to design new topologies from scratch.

Today, the process of designing topologies is decidedly manual: a network architect must invent a set of candidate topologies, and then try to identify the best. An architect typically begins by evaluating the utility and feasibility of topologies described in literature or already deployed in production. This involves quantifying a topology's bisection bandwidth, resiliency to failure, cost, complexity, scalability, and other metrics. An architect then evaluates minor variations of the topology, including changes to connectivity, infrastructure, and scale. This tedious process of modification and evaluation severely limits the number of topologies that can be explored, leading to heavy reliance on intuition. It is our belief that this has caused architects to overlook superior topologies, such as F10 [25], for many years.

In this paper, we describe *Condor*, a pipeline that facilitates the rapid generation and evaluation of candidate topologies, enabling an architect to aggressively explore the topology design space. In Condor, architects express topology designs through a declarative language, which drives a constraint-based generation system. They can then evaluate the utility of multiple design options for an expected workload, using Condor's analysis framework. This framework facilitates the evaluation and tuning of a topology design.

Condor has several objectives. We want to make it easier for network architects to explore a complex design space,

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SIGCOMM '15 August 17-21, 2015, London, United Kingdom

© 2015 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-3542-3/15/08.

DOI: http://dx.doi.org/10.1145/2785956.2787476

thus making it easier to discover novel approaches (e.g., the novel "stripings" we describe in §7) or to rapidly try out new intuitions. We also want to make it easier to express, analyze, and refine specific topologies, while understanding the impact of design choices on tradeoffs between metrics such as bandwidth, fault tolerance, and routing convergence. Our approach is especially valuable for designers of large-scale data-center networks, because of the complexity that arises at larger scales, but it should also be useful to many organizations with medium-scale networks.

We make these contributions:

A declarative language for expressing network topology requirements and goals: Instead of asking architects to design topologies by hand or to implement special-purpose topology-generation software, Condor allows an architect to express requirements and goals using a concise, understandable, declarative Topology Description Language (TDL, §5). Architects use TDL to express high-level properties, rather than specifying low-level wiring.

A constraint-based synthesizer for generating candidate topologies: We implemented a topology synthesizer, using an off-the-shelf constraint solver, to generate specific topologies that comply with TDL descriptions (§6).

Support for network expansion: Operators of large networks typically do not deploy an entire network infrastructure at once, but instead expand their networks in increments. Designing networks to be expandable and designing the phases of an expansion are both challenging problems. Condor is designed to help architects navigate these challenges, both through metrics that evaluate the "expandability" of a network and through TDL and synthesizer support for describing expandable networks and planning the expansion phases – including the problem of expanding a "live" network, while it continues to carry traffic (§9).

2. MOTIVATION

Ideally, a network architect would have a tool that converts a set of requirements and a list of available components into the best possible network design.

Such a tool does not exist. Instead, architects must use an iterative process, generating design options based on their experience and intuition, then analyzing these options against a set of metrics, and refining their designs until they are satisfied. The design space, however, is immense, and architects cannot explore much of it in finite time. We therefore seek a way to speed up this iterative process, while avoiding the exploration of unpromising options.

People have developed tools that assist with this exploration. Mudigonda *et al.* [28] designed tools to find good cost vs. bandwidth tradeoffs for HyperX topologies [1] and for Extended Generalized Fat Trees (EGFTs) [29]. However, these tools are not general: they created different algorithms for HyperX and EGFTs; the algorithms required complex heuristics to reduce the search space; and they analyzed a limited set of metrics (cost, bisection bandwidth, and hop-count). Also, their compute time for even a relatively modest network (8K servers) starts to grow rapidly. Solnushkin [34]

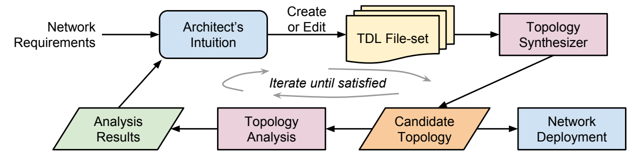

Figure 1: Condor's design iteration loop

presents an algorithm for automated design of 2-layer fat-trees, which improves in some ways on Mudigonda *et al.* [28], but is limited to one kind of tree, and has unknown scaling properties.

Dieter and Dietz [11] described the Cluster Design Rules (CDR) system, which finds an optimal HPC cluster design given a set of requirements. It chooses among a number of network designs (originally not including fat-trees, although this capability was added later [10]), but it does not directly allow specification of network properties, including reliability, and it appears to have hard-coded strategies for network topologies. We could not discover its scalability.

Thus, the current state of the art leads to a relatively long cycle time for iterations that explore novel areas of the design space: someone has to write and debug complex, design-specific, procedural code.

We design the Condor pipeline to *radically decrease the cycle time for this iterative process* and to avoid the need to explore poor tradeoffs. As shown in Figure 1, the architect starts with an intuition about a fabric design, expresses that intuition in a high-level *Topology Description Language* (TDL, §5), invokes a topology synthesizer to generate a set of candidate topologies (§6), and runs these candidates through a suite of topology-analysis functions (§8), to choose the best option.

The architect may have to iterate several times, developing better intuition on each cycle, but Condor does not require anyone to write new topology-specific procedural code, and (as we show in §6.4), Condor's synthesizer runs in minutes or less. Therefore, the architect can iterate rapidly, while optimizing for a combination of many metrics at once.

2.1 The Need for Lifecycle Management

Large datacenter networks often grow to accommodate increasing workloads and evolve to incorporate new technologies. The owner of such a network needs to manage its *lifecycle*, including operations that reconfigure the topology, such as expansions and switch upgrades.

Network operators may need to perform these operations on live networks, without reducing capacity below the network's defined Service Level Objective (SLO) during any operation. It can be difficult to meet this goal, especially given a background rate of component failures [13], the risks inherent in making any changes, and the need to split a major update operation into smaller stages, so as to maintain SLO compliance.

Condor's pipeline addresses two specific aspects of lifecycle management: designing a topology that facilitates future expansions and upgrades, and choosing how to split a network-wide expansion into a set of smaller, SLO-compliant

stages. We postpone discussion of these aspects until §9, so as to avoid complicating our initial description of Condor.

3. TERMINOLOGY

We introduce some terms that we will use in this paper:

Topology Description Language (TDL): A human-readable language that enables architects to declaratively express the high-level properties of a topology's design.

Model: A concrete specification of a topology, including physical objects (e.g., switches, links, racks), abstract objects (e.g., link aggregation groups, pods, power domains), and relationships (e.g., "pod P contains switches $S_1, S_2, ..., S_n$" or "switch S_3 contains linecards LC_1 and LC_2"). Condor's **Synthesizer** converts TDL to models, which can then be evaluated and potentially deployed by network operators.

Striping: The pattern of links inter-connecting components in a topology. In a multi-path tree, the striping determines how uplinks from components at layer L_i are spread across switches at layer L_{i+1} [40]. For recursive topologies, the striping determines how links are distributed between groups of components.

Drain: The administrative removal of a link or component from service. Draining a component drains all child components (e.g., draining a rack drains all switches in that rack).

Failure Domain: The set of components that can fail simultaneously. **Power Domain**: The set of components that could fail due to a fault in one power-delivery component.

4. FABRIC DESIGN FACTORS

Prior work has exposed a vast design space, including fat-trees [3], F10 [25], Quasi fat-trees [43], Aspen trees [40], Jellyfish [33], and many others. Ideally, an architect could simply pick a candidate type from this library, based on an understanding of each design's relative strengths and weaknesses, then choose parameters to support a specific use case.

This still leaves architects with many degrees of freedom. First, they may need to modify one of the canonical designs to reduce costs or to meet other requirements – for example, they can create over-subscribed fat-trees.[1] Second,the architect may have to choose a striping out of a large possible space.

In order to choose between the many available options, an architect must consider many constraints and objectives that apply to datacenter networks. In this section, we discuss the most important of these.

Capital, installation, and energy costs: Architects must consider how much it costs to build a network; "bandwidth at any cost" is not a prudent business model. Prior work [11, 28, 31, 34] has covered cost-optimization; we omit further discussion of such costs for reasons of space.

Bandwidth and latency objectives: The efficiency of many applications is dependent on low-loss and low-latency communication with applications and services located on other machines. Architects must ensure that sufficient interconnection bandwidth is available to minimize the potential for congestion and minimize the path length between machines when possible. However, acceptable bounds on both loss and latency depend heavily on the application workload; two different workloads may vary radically in terms of their tolerance for both loss and latency.

Conceptual complexity: Even if a design is technologically feasible and cost-effective, operators may object to supporting it. For instance, Walraed-Sullivan *et al.* noted [39] that network operators were unwilling to work with random rack-level topologies, such as Jellyfish [33], because it would have been too hard to maintain a human-readable map of the network. Some fat-tree designs also create conceptual complexity for operators and installers; two stripings that are isomorphic may still differ in how easy they are to visualize.

Reliability: While power, cooling, and other critical datacenter infrastructure are often $N + 1$–redundant, it is seldom cost-effective to build $N + 1$–redundant networks. Instead, architects design topologies that gracefully degrade when faced with inevitable component failures. Some prior work has defined reliability in terms of the absence of a partition [8, 9, 39], but an architect may not only want to avoid a partition, but to maintain a given SLO (such as "75% of designed bandwidth") under N component failures.

Some designs, such as F10 [25], are inherently more resilient than others. Alternatively, an architect may choose to add spare capacity to a design, to increase reliability. These choices have costs, so an architect must use a specific failure model, such as MTBF and MTTR for various components, to understand if a design is sufficiently fault-tolerant.

Routing convergence time: Some designs manage routing with distributed routing protocols [4, 7] (e.g., OSPF/BGP); others use centralized SDN controllers [32]. For all such mechanisms, the time for the network to converge to correct routing (loop-free and black-hole-free) depends on the underlying topology. In particular, the topology design determines the number of devices that routing-related information needs to be distributed to, and/or the number of *hops* this information must propagate. We discuss both metrics in §8.

Expandability: As we discussed in §2.1, network owners often need to expand a fabric after installation, sometimes more than once – and may need to do this while the network carries traffic at or near its SLO.

Architects can design fabrics expandable in cost-effective increments [4] that minimize (as much as possible) operational costs while maximizing the chances of maintaining SLOs, even in the face of unplanned component failures during the expansion operation. One may also need to create designs that represent intermediate steps in an expansion, if a single-step expansion would inherently violate an SLO.

Other issues, including physical constraints and compatibility with routing protocols, may also apply; space does not permit us to cover these.

5. TOPOLOGY DESCRIPTION

An architect expresses a network design using Condor's *Topology Description Language* (TDL). TDL is a declarative language that is embedded in an object-oriented language,

[1]Some people reserve the term "fat-tree" for non-blocking fabrics [2]; we use the term in its looser sense.

such as Python or C++, as an Embedded Domain-Specific Language. TDL is "declarative" in the sense that it is not used to procedurally define a concrete network topology. Instead, architects use TDL to express a topology's structural building blocks and the relationships between them (§5.1), along with potential connectivity (§5.2.1), and constraints on connectivity (§5.2.2). Condor's synthesizer (§6) converts TDL into a set of candidate *models* (§3), each of which represents a concrete topology that satisfies the constraints declared in TDL. Thus, Condor allows an architect to focus on defining the characteristics of the desired topology, rather than on writing procedural code to explore the design space.

TDL is most similar to NED [38], a language incorporated into the OMNeT++ [37] network simulation framework. However, an architect uses NED to describe a *concrete* topology, and must procedurally define the connectivity among building blocks.

A given TDL description specifies a (possibly empty) set of candidates, rather than exactly one, because there is no *a priori* reason to assume that exactly one topology satisfies the chosen constraints. We expect an architect will explore the design space by adding and removing constraints, and then applying the analysis suite (§8) to decide which, if any, of the candidates meets their requirements. This iterative process is required because we do not know how to express all of an architect's high-level objectives, such as SLO compliance, as constraints on connectivity. While we expect architects to examine the models (via a visualization tool) to gain insight into the candidates, they never directly edit these models. If they need to make changes, they must modify the building blocks, constraints, and other details in the TDL description, and synthesize new candidates. Separating a topology's description from the procedural code required for generating a model makes it easier to capture and understand an architect's intent, dramatically reducing the potential for errors.

TDL can be spread across multiple files, known as a *TDL file-set*, which in our current system consists of one or more Python modules. The use of multiple files supports modularity and reuse. In addition, TDL makes it easy to re-use common aspects of network designs: not just building blocks, but especially the essential aspects of a "species" of related fabrics. That is, an architect can start with a generic species (e.g, a fat-tree built from 48-port switches) and then include additional constraints to create a concrete "individual" network. We designed the TDL to support a wide range of topology species, including tree [3, 25], Quasi fat-tree [43], recursive [15, 16], some flattened-butterfly topologies [22], Quartz [26], Jellyfish [33], and others.

We show how TDL can be used to express a fat-tree topology [3] in Figure 2. We will refer to this example throughout our discussion of TDL constructs.

```
1  # reusable 10GbE switch building block
2  class Switch10GbE extends TDLSwitchBlock:
3    function Switch10GbE(num_ports, name):
4      port = new TDLPortBlock(TENGBPS)
5      Contains(port, num_ports)
6      ...
7  # parameterizable FatTree building block
8  class FatTree extends TDLBuildingBlock:
9    function FatTree(num_pods):
10     # equations from Alfares et al. 2008
11     num_ports = num_pods
12     num_spines = pow(num_pods / 2, 2)
13     num_sw_per_pod_tier = num_pods / 2
14
15     # spine, agg, & ToR switches
16     spine = new Switch10GbE(num_ports, "spine")
17     agg = new Switch10GbE(num_ports, "agg")
18     tor = new Switch10GbE(num_ports, "tor")
19
20     # a pod contains agg and ToR switches
21     pod = new TDLBuildingBlock("pod")
22     pod.Contains(agg, num_sw_per_pod_tier)
23     pod.Contains(tor, num_sw_per_pod_tier)
24
25     # a fat-tree contains spines and pods
26     Contains(spine, num_spines)
27     Contains(pod, num_pods)
28
29     # pairs of components eligible for connection
30     pod_connector = pod.Connects(agg, tor)
31     s_connector = Connects(spine, agg)
32
33     # constraints on the Connector objects
34     # in each pod, an agg connects to every ToR
35     pod_connector.ConnectPairsWithXLinks(agg, tor, 1)
36
37     # every spine connects to every pod
38     s_connector.ConnectPairsWithXLinks(spine, pod, 1)
```

Figure 2: TDL pseudocode describing a fat-tree topology

```
1  # Base class for all TDL blocks
2  class TDLBuildingBlock:
3    function TDLBuildingBlock(name):
4      ...
5    function Contains(bb_obj, num_of):
6      # Records parent-child relationship
7      ...
8    function Connects(lhs_bb_obj, rhs_bb_obj):
9      # Records and returns new TDLConnector obj
10     ...
11 # Specialized TDL classes
12 ## L2/L3 Device
13 class TDLSwitchBlock extends TDLBuildingBlock
14 ## L2/L3 Device Port
15 class TDLPortBlock extends TDLBuildingBlock
```

Figure 3: Pseudocode describing TDL building block classes

5.1 Describing Building Blocks

An architect begins by describing a topology's common *building blocks*, each representing a physical component, such as a switch, patch panel, or rack, or a logical component, such as a "pod" in a fat-tree topology [3]. The architect can also describe parent-child ("contains") relationships between building blocks. These blocks and the relationships between them form an abstract tree, with one block designated as the root upon instantiation.

Figure 3 sketches the classes used to describe building blocks in TDL, including specialized classes used to distinguish physical devices (like switches and ports) involved in connectivity. Building blocks can be described by instantiating an instance of one of these classes. In addition, because TDL is embedded in an object-oriented language, architects can generate a library of reusable, parameterizable building block classes by declaring subclasses. These subclasses can be then be reused and shared among topology descriptions.

Building blocks are a natural abstraction for topology description. For instance, a homogeneous fat-tree network constructs spine (called "core" in [3]), pod, aggregation, and top-of-rack (ToR) logical building blocks, all from one physical switch type.

Our corresponding fat-tree example (Figure 2) begins with the declaration of a parameterizable building block class,

Switch10GbE. This building block contains a variable number of ports, and is instantiated to describe spine, aggregation, and ToR switches in the FatTree building block class. During instantiation, a name describes the purpose of each switch block instance.

The parameterizable *FatTree* building block class starts at line 8. An instance of this class serves as the root of the abstract tree, and contains spine and pod building blocks. The pod building block (lines 21 - 23) in turn contains aggregation and ToR switch building blocks (instantiated on lines 17 - 18). Each switch building block object in the FatTree block is instantiated from the Switch10GbE building block class.

This hierarchical model also works for describing recursive topologies, such as DCell [16], a topology that recursively uses two physical building blocks (switches and servers) and a logical building block for each level of aggregation. In a DCell topology, level DCell_0 contains a switch and multiple endhosts, while a $\text{DCell}_{i>0}$ contains multiple DCell_{i-1}s.

In addition to describing hierarchy, the architect can record meta-information, such as the failure properties (Mean Time Between Failures, Mean Time to Recovery) of a physical device (used in §8.4).

5.1.1 Grouping by Rule

TDL supports rules to express *groups* of components, in cases where they cannot easily be described with a "Contains" relationship. For instance, after describing a fat-tree's physical and logical network components, an architect can use rules to describe how each physical component is assigned to a rack, to a failure domain, or to an arbitrary group. As with connectivity constraints, architects can quickly modify grouping rules to try different assignment approaches or to handle variations between physical facilities. We use rules to generate groups in §7, when describing F10 with TDL.

5.2 Describing Connectivity

Given a set of building blocks, the architect describes how they should be connected, by defining candidate connections with *connectivity pairs*, and then using *constraints* together with *tiebreakers* to eliminate potential connections.

5.2.1 Defining Connectivity Pairs

Connectivity pairs express pairs of components and/or groups that are eligible to be connected. A connectivity pair consists of left-hand and right-hand side (LHS and RHS) building block objects, and is expressed by calling "Connects" on a "scope" building block. The scope building block must "Contain" both of the building blocks in the connectivity pair. During synthesis, for each scope component instantiated from the scope building block, the set of candidate connections is equal to the Cartesian product of the component's successors that were instantiated from the LHS and RHS building blocks, respectively.

For instance, in lines 30 and 31 of our fat-tree example (Figure 2), the architect defines two connectivity pairs, each specifying a pair of building blocks. Abstractly:

1. In every pod, connect aggregation & ToR switches
2. In every fat-tree, connect spine & aggregation switches

```
1  class TDLConnector:
2    ...
3    # constraints on connectivity of objects
4    # derived from a specified building block
5    function ConnectXLinks(bb_obj, x_links)
6    function ConnectAtMostXLinks(...)
7    ...
8    # constraints on pairwise connectivity
9    function ConnectPairsWithXLinks(bb_obj_1, bb_obj_2, x_links)
10   function ConnectPairsWithAtLeastXLinks(...)
11   ...
12   # support for custom constraints
13   function AddConstraint(constraint_obj)
```

Figure 4: Partial set of TDL constraints

The *In every* part defines the scope of candidate connections. The first connectivity pair, for example, ensures that a ToR switch can only be connected to an aggregation switch in the same pod. Thus, for each pod, the set of candidate connections is the Cartesian product of the pod's aggregation and ToR switches. Similarly, the second connectivity pair permits connections between spine and aggregation switches in the same fat-tree.

For some topologies, such as flattened butterfly [22] and recursive topologies, some connectivity pairs have the same building blocks as the LHS and RHS. For example, for a DCell network, the connectivity pairs are:

1. In every DCell_i where i > 0, connect all DCell_{i-1}s.
2. In every DCell_i where i = 0, connect hosts and switches.

Connectivity pairs can be declared at different *levels* of the hierarchy. In DCell, the pairs could be defined between hosts, instead of between DCell_{i-1}s, which would enable an architect to express constraints at finer granularities. However, it also increases the number of pairings produced by the Cartesian product, expanding the solution space that the synthesizer must navigate. We would need this level of granularity to generate topologies that exactly match those in the DCell paper [16], but not to meet the paper's DCell specification, as discussed in §5.3.

5.2.2 Constraints and Tiebreakers

After defining candidate connections, the architect specifies constraints on connectivity. Without such constraints, the synthesizer will simply make arbitrary connections between eligible components until connections are no longer possible (e.g., no more ports, etc.). This, for example, could leave a single ToR switch connected to a single aggregation switch, creating a single point of failure or a disconnected network. Constraints and tiebreakers allow the architect to guide the synthesis process to avoid bad choices.

Constraints: Constraints can be expressed on the building blocks referenced within a connectivity pair or any of their predecessors, and can be specified on individual building blocks, or on pairs or groups of building blocks. Constraints are associated with a TDLConnector object that is returned when a connectivity pair is declared, and constrain the set of candidate connections in the connectivity pair's scope. Figure 4 shows a partial set of TDL constraints.

Constraints applied to an individual building block can set the minimum, maximum, or exact number of connections for every component instantiated from that building block. Every component starts with a default constraint, to ensure that the

component never has more connections than available ports. Constraints propagate downwards; e.g., if a switch containing linecards cannot have more than N connections, then the sum of its linecards' connections can not exceed N.

Constraints applied to a pair of building blocks bound the number of connections for every pair of associated components, or between a single component of the LHS descriptor and all components of the RHS descriptor.

An architect's choice of pair-wise constraints is the most important tool for balancing the many tradeoffs in topology design, and hence represents the architect's primary design insights. In particular, an architect can use pair-wise connectivity constraints to achieve fault-tolerance through diversity.

In lines 35 and 38 of our fat-tree example (Figure 2), we use pair-wise constraints to describe a fat-tree's connectivity:

1. Every aggregation switch must connect to every ToR.[2]
2. Every spine switch must connect to every pod.

By default, a solution must satisfy the boolean AND of all per-component and pair-wise constraints. However, TDL supports explicit ORing of constraints (OR operations can be recursive), which is especially useful for pair-wise constraints: e.g., a component pair must have either X connections or 0.

Tiebreakers: Condor's synthesizer (§6) uses a constraint solver to find candidate topologies that meet the constraints in a TDL description. The constraint solver iterates serially through the candidate connections when finding things to connect. Sometimes, but not always, the order of this iteration is key to finding a solution efficiently, or affects a property that matters to the architect, or both (see §9.4).

An architect can include an ordered set of one or more *tiebreakers*, written as callback functions, to control this ordering. (We include a default tiebreaker to maximize human-readability of the generated connectivity.)

Late-bound constraints: An architect may wish to apply constraints whose numeric parameters vary based on decisions made during synthesis. *Late-bound constraints* (LBCs) are pre-defined in TDL as alternatives to architect-written procedures to derive such constraints, and are compiled into min/max pair-wise constraints during synthesis. The only LBC currently defined is "diversity." This LBC spreads connectivity across as many pairs as possible, and ensures the number of connections between pairs is equal ± 1.

5.2.3 Custom and group constraints

Custom constraints: The synthesizer's solver supports more complex constraints than described so far. Since TDL is embedded in an object-oriented language, architects can write custom constraints that interact with the solver by subclassing the *Constraint* class and then defining their constraint. For instance, one could define an *oversubscription ratio* constraint for a tree, bounding the ratio between the number of *incoming* links from a lower stage and the number of *outgoing* links towards an upper stage. We show in §7 how we use custom constraints to express designs such as F10 [25].

Constraints on groups: Connectivity constraints can refer to groups (§5.1.1), so that (e.g.) one can require that every aggregation switch in a fat-tree connects to spine switches in at least two power domains.

As another example, suppose one wants to reserve ports 1–4 on each ToR for connecting to aggregation switches (perhaps to simplify wiring). This could be expressed by adding a new rule to define groups of port objects based on port number, then adding connectivity constraints dictating where these groups can be used.

5.3 Examples of TDL's Concision

We believe (but cannot prove) that TDL's concise descriptions will lead to better designs and fewer errors. To support this, we counted lines of code for TDL descriptions of several networks. As shown in Figure 2, TDL can express all of the fat-tree designs in Al-Fares *et al.* [3] in under 30 non-comment lines. TDL can express BCube networks in 32 lines and Facebook's fabric [4] in 33 lines. To express DCell-like networks,[3] we need only change 2 lines from BCube's description, and add 10 more.

6. NETWORK SYNTHESIS

Condor's synthesizer converts a TDL file-set into a model for a network topology that complies with the TDL specification. A model represents both the parent-child relationships between components (the *hierarchy graph*, §6.1) and the connectivity between components (the *connectivity graph*, §6.2-§6.3). Synthesis of connectivity involves solving three problems: formulating synthesis as a constraint satisfaction problem [36] (§6.2); allowing an architect to influence the results via tiebreakers (§6.2.1); and optimizing the synthesizer's performance to support a rapid design cycle (§6.3).

The synthesizer may fail to generate a model, either because the constraints in the TDL description cannot be satisfied or because finding a suitable result exceeds a timeout. In such cases, the architect may have to try again with different constraints or tiebreakers. If more than one model would satisfy the constraints, the synthesizer generates only one; the architect can generate additional candidate models by adding, removing, or changing the tiebreakers (§6.2.1).

6.1 Synthesizing the Hierarchy Graph

The synthesizer constructs a hierarchy graph, beginning by converting the root building block instance into the graph's root "component," and then recursively converting contained building blocks (expressed with the "Contains" relationship in TDL) into successor components. Each component represents a concrete incarnation of a building block. For instance, in our fat-tree example (Figure 2) the synthesizer generates

[2]Constraints are applied to the set of candidate connections generated during synthesis for each scope; thus, this constraint only requires connections between aggregation and ToR switches in the *same* pod.

[3]It would take more lines of TDL and more solver memory to synthesize the exact striping defined in the DCell paper. The striping we produce meets all of the specifications defined in the paper, and only requires a trivial adjustment to DCell's path lookup algorithm.

a component for the root building block (FatTree). Then, since the root building block contains pod and spine building blocks, the synthesizer instantiates one or more pod and spine components. This recursion continues, resulting in the subsequent instantiation of aggregation, ToR, and port components.

After the synthesizer instantiates a component, it adds a reference from the component back to the building block that it was derived from, and adds a directed edge in the hierarchy graph from the parent component to the child component (i.e., each pod will have a directed edge to each of its ToR switches). Once the synthesizer has instantiated all components, it generates groups based on the grouping rules defined in the TDL (§5.1.1). The synthesizer adds directed edges from each group object to the components it contains.

6.2 Synthesizing Connectivity

Most of the work of synthesis is to find appropriate connectivity between components in the hierarchy graph. The synthesizer generates connectivity by processing the connectivity pairs associated with components in the graph. Components are processed in reverse of the order in which they were instantiated. Thus, connectivity generation begins at the *lowest* non-leaf level or stage in the hierarchy graph (e.g., at the aggregation and ToR components in our fat-tree example).[4]

For each component, the synthesizer identifies the associated building block and determines if the building block references connector objects (generated by the use of the "Connects" construct in TDL). Each connector object is processed independently during synthesis. The synthesizer generates a set of candidate connections and then applies constraints and tiebreakers to choose the connections to add. The chosen connections are then recorded as edges in the connectivity graph, and the process repeats for the next connector and for subsequent components.

Scope: For each component with a connectivity pair, the synthesizer will only consider connections within the component's *scope*, defined as the component's successors in the hierarchy graph. Thus, when processing the connectivity pair associated with a pod in our fat-tree example (Figure 2), the synthesizer will only consider connections between the ToR and aggregation switches within that pod.

Generating candidate connections: For each connector, the synthesizer creates two lists of components in the hierarchy model. Each list contains the in-scope components that correspond to one of the two building blocks referenced in the connector's connectivity pair. The Cartesian product of these two lists (after removing any self-connections) represents all possible *candidate connections* between the components within them. For instance, when the synthesizer processes a pod component in our fat-tree example, it will generate a list of aggregation and ToR switches that are successors of the pod in the hierarchy graph. Every pairing of an aggregation and ToR switch will be considered a candidate connection.

[4]In tree networks, this approach ensures, without the need for additional constraints, that sufficient ports remain available for connectivity at higher levels.

Finding the right connections: Out of the set of candidate connections, it is likely that only a proper subset can be established without violating constraints in the TDL description (although in some cases, constraints may prevent establishment of *any* candidate connections). We treat the problem of determining which connections *should* be made as a constraint satisfaction problem (CSP) [36]. CSPs consist of a set of decision variables, each assigned an integer domain, and a set of constraints on a collection of one or more decision variables. Our implementation uses the open-source *or-tools* constraint solver [14], but other solvers could be used.

Generating decision variables: For every candidate connection, the synthesizer generates an integer variable to represent the number of links that could be established in the current scope for that candidate. This variable has a domain of 0...*MinPairPorts*, where *MinPairPorts* is the minimum, over the two components in the candidate, of the number of successor ports still available for connection, given that some of these components' ports may have already been used.

Following the generation of such an integer variable, the synthesizer stores the variable in a data structure that supports the lookup of candidate connectivity associated with any component, or between any pair of components. This structure also allows retrieval of any variable associated with a component's predecessor, or pair of predecessors.[5] The synthesizer uses this structure to set up constraints.

Constraining decision variables: Next, the synthesizer applies constraints, converting late-bound constraints to numerical constraints based on the model's state.

TDL enables architects to describe constraints on the connectivity of components derived from a building block, or between pairs of components derived from a pair of building blocks. However, these high-level constraints cannot be processed directly by the underlying constraint solver. Instead, the synthesizer must convert these high-level constraints into integer expressions, and then compose constraints from these integer expressions, combined with boolean and relational operators. The synthesizer then provides these constraints to the solver to guide the generation of connectivity.

Custom architect-defined constraint classes (§5.2.3) have a similar workflow. These custom TDL constraints, implemented as callbacks, retrieve the variables associated with one or more components and/or component pairs, then generate and return a set of CSP constraints, which the synthesizer passes to the solver.

6.2.1 *Navigating the Solution Space*

Following the generation of decision variables and the application of constraints, a solution is requested from the constraint solver. The solver operates by choosing a variable from a set of unbound decision variables and assigning it a value within its domain. The solver stops once it has assigned a value to all variables or determines that a solution is infeasible.

[5]This association is important when processing constraints, as a candidate connection between components C_1 and C_2 is also a candidate connection between their predecessors.

Tiebreakers: At the beginning of each decision, the solver must chose a decision variable to operate on next, from a set of unbound variables. By default, the relative rank of these decision variables is equal, and thus unbound variables are processed in the order they were instantiated. The order of this iteration determines which valid solution is returned,[6] and is sometimes key to efficient synthesis (see §9.4).

Architect-defined tiebreakers (§5.2.2), implemented as callbacks, impose a rank ordering on the set of unbound decision variables. When the solver needs to choose the next variable to operate on, it updates the rank of decision variables by serially invoking these callbacks (if any).

6.2.2 *Processing the Solver's Output*

If the solver finds a solution for a scope, it binds each integer variable instantiated for that scope to a single value within its domain, which represents the number of links connecting a pair of components. For each such link, the synthesizer allocates an available successor port[7] from each of the connected components and generates a new link object to represent the connection between these ports. These link objects, taken together, form the connectivity graph. When two components are connected via multiple links, we also add an aggregated logical link to the graph, which simplifies later analysis (§8).

The solver may fail to find a solution for one of several reasons: the constraints are unsatisfiable, or the solver has run longer than a user-specified timeout, or the solver has had to backtrack more than a user-specified number of times. Depending on the reason for failure, the architect might need to change the constraints or improve the tiebreakers in order for the solver to find a solution.

6.3 Efficient Synthesis

Avoiding backtracking during synthesis: Constraint solvers search for a solution by selecting an unbound variable and assigning the variable a value in its domain. Each assignment constrains the solution space and, depending on constraints, may reduce the domains of unbound variables (potentially to the point of forcing an assignment).

Constraint solvers are general purpose tools and are oblivious to our problem domain. Without guidance, a constraint solver may assign values to variables in a manner that constricts the search space to the point that the solver cannot find a valid solution, even when solution is possible. For instance, the *or-tools* constraint solver used by Condor will (by default) assign a variable the minimum value in its domain, thus setting the number of connections between every pair as 0. This naïve approach will almost certainly lead to *backtracking*, a state where the constraint solver detects that a solution is infeasible (given previous assignments) and reverts one or more of its previous decisions. By default, a constraint solver will branch and backtrack until it finds a solution or has explored the entire search space; both backtracking and full search can be costly (see §9.4 for an example where exploring the entire search space is infeasible).

To facilitate a rapid design loop, the synthesizer allows the user to set limits on the number of branching and backtracking operations, and on the maximum time allowed for exploring the solution space. We have also designed heuristics that reduce the potential for backtracking, including (among others, not described due to space constraints):

Greedy connection assignment: We use a "greedy" heuristic to tell the solver to assign as many connections as possible, at each step, to each integer variable; this tends to minimize backtracking. Being greedy also allows the solver to spread connections across a smaller set of components, which is critical when synthesizing expandable topologies (§9).

Efficiently synthesizing recursive topologies: The synthesizer must allocate an integer variable for every candidate connection, so synthesis of large, direct-connect topologies could require millions or even billions of such variables. For instance, synthesizing a DCell_3 [16] network with N=4, described with host-to-host connectivity, would require over half a billion variables. Memory limits often make solving this size of constraint problem infeasible, and, even if sufficient memory is available, solving a CSP with millions of variables can take a long time. Instead, we leverage the recursive nature of these topologies to reduce the number of decision variables required.

For example, in §5.2.1, we defined the connectivity of a DCell topology at the granularity of DCell building blocks ("In every DCell_i where i > 1, connect DCell_{i-1}s"), instead of at the granularity of individual server building blocks. For many recursive topologies, including DCell, it is possible to describe the topology at this level of granularity, as long as links generated between "DCells" are balanced evenly across the end hosts within the group; TDL can express this balance.

6.4 Synthesizer Performance

Table 1 shows synthesizer CPU and memory costs for a variety of networks. We use the NetworkX graph library [17] for storing all graph structures as adjacency lists. NetworkX is written in Python and could probably be replaced with something much faster and more memory-efficient. Our current implementation of the rest of the synthesizer is also in Python, which adds some overhead; we plan to implement a new version in C++. However, the current performance is already fast enough for most purposes, and we see approximately linear CPU and memory scaling (vs. the number of end hosts) within a family of topologies.

7. THE POWER OF TDL: TO F10 AND BEYOND!

In this section, we show that the TDL allows an architect to rapidly discover novel stripings in a fat-tree [3] topology.

The traditional fat-tree striping, shown in Figure 5, can be expressed as: the Nth aggregation switch in every pod connects to the same spine switches, and every spine switch

[6]Because an architect expresses connectivity declaratively with pairs and constraints, rather than by procedurally specifying connections between individual components, the constraint solver could return any one of multiple valid solutions.

[7]If a building block has multiple types of successor port (e.g., 10Gb and 40Gb ports), the architect must specify a choice, in TDL, as an attribute of the relevant connectivity pair.

Topology	Switch count	Link count	# of end hosts	CPU secs‡	Mem. (GB)
DCell, n=16,k=2	4641	148512	74256	75	2.59
DCell, n=24,k=2	15025	721200	360600	346	12.5
BCube, n=16,k=3	16384	262144	65536	116	4.0
BCube, n=24,k=3	55296	1327104	331766	597	20.9
FatTree*, 16 pods	320	2048	1024	2	0.069
FatTree, 40 pods	2000	32000	16000	16	0.60
FatTree, 64 pods	5120	131072	65536	61	2.4
FatTree, 80 pods	8000	256000	128000	117	4.8
Facebook Fabric†	5184	36864	221184	57	2.5

*** all fat-trees are 3-stage ‡no trials here needed backtracking**
† assuming this uses 96 pods, 96-port switches, 4 uplinks/ToR

Table 1: Synthesizer performance for various examples

connects to every pod once. Given this striping, if aggregation switch N in pod X fails, none of the Nth aggregation switches in *any* other pod can reach pod X. As discussed in §8.3, such a failure results in packet loss until the network's routing has reconverged.

Describing F10: In F10, Liu *et. al* [25] reduce the impact of an aggregation switch failure on the connectivity between two groups of pods. With F10, pods are connected to spine switches using either of two different stripings (A or B), as shown in Figure 5. The use of two stripings ensures that the Nth aggregation switch in a pod with striping A is able to direct traffic to a destination in a pod with striping B through multiple disjoint (shortest) paths.[8]

Although the F10 authors described the construction of the A and B stripings arithmetically, we use the following constraint to describe their construction: *Where* agg_A *and* agg_B *are aggregation switches in pods with A and B stripings, respectively, every pair of aggregation switches* (agg_A, agg_B) *can connect to each other via at most one spine switch.* This constraint ensures that an aggregation switch in a pod with striping A connects to as many different aggregation switches in a pod with striping B as possible.

Using TDL, we can express a fat-tree with A and B stripings by adding constraints to a traditional fat-tree:

1. We define rules to split pods equally into two groups, A and B (§5.1.1).
2. We define a custom constraint to prohibit any pair of aggregation switches (a, b) (with $a \in A$ and $b \in B$) from from having more than one spine switch in common.

With TDL, we express the difference between fat-trees and F10 in 29 lines (including 18 lines for the custom constraint), much less code than a procedural Python program to model a variable-size AB fat-tree (about 150 lines).

Beyond F10: With TDL, we were able to compartmentalize F10-specific constraints from the basic fat-tree constraints. In this section, we discuss how this modularity led to new insights.

Once we understood how F10's AB stripings could be expressed with TDL, we wondered if we could further reduce the impact of an aggregation switch failure by adding an additional striping. By modifying the TDL grouping rule

[8]The number of disjoint paths increases with topology size.

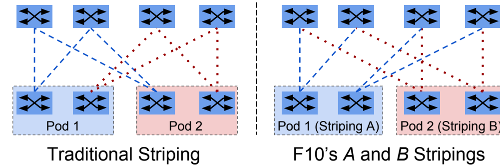

Figure 5: Connectivity with traditional and F10 fat-tree stripings

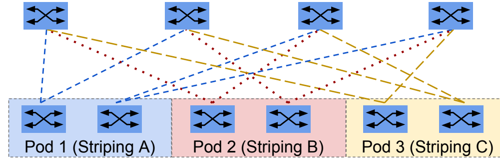

Figure 6: Connectivity with A, B, and C stripings

to split pods into three groups (A, B, C) and applying our F10-inspired constraint to all pairs of groups, we generated the ABC topology shown in Figure 6. Like F10's A and B stripings, every aggregation switch in a pod with striping S has 2+ disjoint paths to a pod with a striping other than S.[8] However, using three stripings increases the probability of any pair of pods having a different striping to 66%.

After generating the ABC stripings, we investigated the bounds on the number of stripings we could create. Using the same TDL, we synthesized larger fat-tree topologies (5+ pods), varying the number of groups that P pods are divided into between 1 and P. Our F10-inspired constraint requires that each group has a distinct striping (e.g., splitting pods into three distinct groups requires three distinct stripings: A, B, and C).

From this experiment, we determined that the number of possible stripings grows sub-linearly with the number of pods. For instance, we can generate up to 4 stripings (A, B, C, and D) for a 6-pod fat-tree, and up to 6 stripings for a 10-pod fat-tree.[9]

We extracted the following intuition from our analysis of the generated stripings and constraints:

- Our constraint specifies that any pair of aggregation switches from pods with different stripings may connect to at most one spine switch in common.
- Let $SpineSet_i$ and $SpineSet_j$ be the sets of spine switches connected to aggregation switches agg_i and agg_j, respectively. From our constraint, for any agg_i and agg_j in pods with different stripings, $SpineSet_i \cap SpineSet_j \leq 1$.
- By construction, *NumUplinksPerAgg* and *NumAggsPerPod* are equal in a fat-tree [3], and their product is equal to *NumSpines*. Thus, for any agg_i and agg_j in different stripings, $SpineSet_i \cap SpineSet_j = 1$.
- Let $AggSet_j$ be a set of aggregation switches from pods with different stripings connected to $spine_j$ ($|AggSet_j| = NumStripings$). Given our constraints, $\forall\ agg_i, agg_k \in AggSet_j$, s.t. $i \neq k$, $SpineSet_i \cap SpineSet_k = \{spine_j\}$, meaning each agg_i needs a disjoint $SpineSet_i \setminus \{spine_j\}$. The maximum number of disjoint $SpineSet_i \setminus \{spine_j\}$ (and thus the maximum value of *NumStripings*) is equal to (*Num-*

[9]These stripings are available at http://nsl.cs.usc.edu/condor.

Spines minus $spine_j$) divided by (*NumUplinksPerAgg* minus the uplink to $spine_j$).

From these, we can define an upper bound on the number of stripings possible for a fat-tree of a given size:

$$NumStripings \leq \frac{NumSpines - 1}{NumUplinksPerAgg - 1} \quad (1)$$

Our experimental results coincide with this upper bound (e.g., we were able to generate up to 6 stripings in a 10-pod fat-tree). However, it becomes computationally difficult to synthesize the maximum number of stripings in larger topologies; Condor's synthesizer cannot generate 12 stripings, the upper bound for a 22-pod fat-tree, within 30 minutes. When we investigated how to improve performance, we determined that our F10-inspired constraint generates *Balanced Incomplete Block Designs* [19, 42] (BIBDs). Efficient synthesis of BIBDs is a known problem [27], and the exact conditions required for their existence are not known [18].

However, in larger topologies where synthesis is more difficult, we can trade improved synthesizer performance for fault-tolerance. Our F10-inspired constraint provides maximum fault-tolerance, by ensuring that every pair of aggregation switches in pods with different stripings can communicate through the spine layer.[10] Depending on the level of fault-tolerance required, our constraint can be adjusted to allow aggregation switches in different pods to connect to between 1 and *NumUplinksPerAgg* − 1 spine switches in common, which respectively varies fault-tolerance between *NumUplinksPerAgg* and 2 disjoint paths. As the level of fault-tolerance guaranteed by the constraint is reduced, the size of the corresponding solution space is increased, and synthesis performance is dramatically improved.

8. TOPOLOGY ANALYSIS

Once a TDL file-set has been synthesized into a model, an architect can analyze the model against various metrics, such as those described in §4. We would like these analyses to be both accurate (to avoid misleading the designer) and fast (to enable rapid iterations).

Some metrics, such as cost, are easily calculated. However, a topology's utility depends largely on its ability to maintain SLO compliance, determined by how traffic patterns interact with the topology throughout its lifecycle. These interactions can either be calculated precisely through detailed simulation, which is quite hard to scale to large datacenter networks, or through faster but approximate algorithms.

For example, we cannot accurately quantify the "bandwidth" of a topology for a given traffic matrix independent from how routing protocols, transport protocols, switch queueing mechanisms, and traffic engineering interact. The likelihood of failed and "drained" components and links adds more complexity.

Papers about datacenter networks typically fall back on proxy metrics, such as bisection bandwidth and hop-count [8, 15, 16, 28, 31, 33]. These metrics can be relatively easy to compute but do not reflect the full complexity of the problem [20], and they ignore the reality that most networks are never operated under worst-case traffic matrices. Also, "[f]inding the exact value of the bisection width has proven to be challenging for some specific networks" [5]. Network owners learn, through hard-won operational experience, that other proxy metrics are better predictors of SLO compliance.

Yet without a means to quantify a topology's bandwidth, we cannot develop good metrics for concepts such as reliability or expandability, since these are best described as a network's ability to maintain its bandwidth SLO in the face of random link failures or planned drains. (Some prior work has used the probability of a topology becoming partitioned as an indicator of fault-tolerance [8, 9, 39], but this generally is not sufficient to predict compliance with a minimum-bandwidth SLO.) Therefore, we use an approximate bandwidth metric (§8.1) to illustrate how changes in the TDL for a network (or its expansion plan) affect several metrics of interest.

We do not claim to have solved the problem of quantifying a topology's bandwidth; our metrics have many limitations. We believe that our metrics provide real value to network architects and hope that they motivate further research.

8.1 Approximate Bandwidth Metric

Our approximate bandwidth metric (ABM) evaluates a topology's ability to support an architect-defined traffic matrix. (We do not assume a worst-case traffic matrix, because many networks are unlikely to be operated under such conditions. Architects often maintain spare or protected capacity and employ intelligent scheduling techniques to improve job locality [21, 23, 30, 41].) The ABM can be used to compare an application's predicted throughput against a set of candidate topologies (e.g., tree topologies with varying levels of over-subscription) and to evaluate application throughput when a topology is in a degraded state (§8.2).

An architect uses historical and (projected) future demands to generate one or more pair-wise traffic matrices of offered loads between components (e.g., host-to-host or ToR-to-ToR).[11] For each traffic matrix, the metric estimates the throughput of each flow, assuming fair sharing of overloaded links. We estimate throughput by propagating flow demands through the network and proportionally reducing each flow's demands at each saturated link. We reduce this to an aggregate scalar: the sum of achieved flow rates divided by the sum of offered loads. (Other aggregates are possible.)

In a multipath network, we must split flows as they propagate through each junction point. For the results later in this paper, we treat flows as fluid[12] and split them at each junction between all shortest-path next-hops. By default, flows are split equally among next-hops, but we also support dividing flows based on the number of physical links to each next-hop (comparable to ECMP). We can extend this metric to approximate WCMP [44] or architect-specified routing functions, but we omit these details for space reasons.

[10] E.g. in a 22-pod fat-tree, an aggregation switch with striping S has 11 disjoint paths to a pod with a striping other than S.

[11] Matrices can be weighted to produce an aggregate score.

[12] We assume flows defined in traffic matrices represent an aggregate of application flows, and thus are fluid and splittable.

Limitations: The ABM only supports tree topologies. In addition, to reduce computation time, we perform two rounds of flow propagation: (1) upwards propagation from ToRs towards higher tiers of the tree and (2) downwards propagation from higher tiers towards ToRs. As a result, our metric only determines a lower bound on the throughput of each flow in the traffic matrix and only works for strict up-down routing. It cannot handle (for example) F10, which uses *ToR bouncing* to route around failures [25].

8.2 Simplified Reliability Metric

Architects need to know the probability that a candidate topology will maintain its bandwidth SLO, given a *failure model* including the Mean Time Between Failures (MTBF) and Mean Time to Repair (MTTR) for each component.

TDL allows specification of MTBF and MTTR for building blocks. We use these values to compute an SLO metric, using Markov Chain Monte Carlo (MCMC) simulation [35]. For each MCMC iteration, we apply the ABM to the topology, treating currently-failed components (and their successors) as missing, and record the bandwidth achieved for each flow.

After a user-specified number of iterations, we compute the fraction of flows that complied with a user-specified SLO. The SLO is stated as a pair of values: a target throughput and a minimum fraction of iterations during which a flow must meet or exceed that target (e.g., "a flow achieves 4Gbps 99.9% of the time").

Limitations: Our simplified reliability metric does not attempt to account for short-term capacity and packet loss that may occur during convergence (§8.3).

8.3 Routing Convergence Metric

When components fail in large networks, it may take some time for the routing system to re-converge. Global re-convergence following a single link or switch failure can require 10s of seconds in a large network, during which time some flows will experience 100% packet loss [40].

Topologies can differ in how rapidly routing can converge. For instance, the stripings described in *Aspen Trees* [40], F10 [25], and §7 can lower or eliminate convergence time in tree topologies after certain failures, by increasing the probability that a switch can handle a failure locally.

We quantify how quickly a topology's routing can converge by calculating two proxy metrics: the number of switching components that routing-related information needs to be distributed to and the number of *hops* this information must propagate. We calculate these metrics in two phases.

In the first phase, for every switching component C, we determine how the failure of any other component affects C's ability to route traffic to destinations (hosts or other networks), subject to the routing algorithm in use (such as strict up-down routing). If C has N paths towards destination D, we compose N sets of "path components," $P_{C,D,1}, \ldots, P_{C,D,N}$, with each set $P_{C,D,i}$ containing all of the components on path i between C and destination D. The intersection of these sets yields a set of components that appears on all N paths. The failure of any component in this set or any of their predecessors will prevent C from routing traffic to destination D. We store this intersection of *Routing Resiliency Limiters* (RRLs) for every (C, D) pair.

In the second phase, we identify which failures a component C can *react* to. For each destination D, component C can react to the failure of any path component F as long as $F \notin RRLs(C, D)$, by making a routing decision. We record each such F as a component whose failure C can react to.

Finally, we compute two metrics: first, the number of components (Cs) that can react when any component F fails; this is useful for SDN networks, as it counts the number of switches needing updates. Second, the maximum distance between C and F over all such pairs; this "routing-info radius" (RIR) can be used to estimate the routing convergence time for distributed routing protocols as it approximates how far F's failure must propagate.

Limitations: Our routing convergence metric does not quantify the amount of *time* required for convergence. Walraed-Sullivan *et al.* [40] used an alternate approach, simulating specific routing protocols to estimate convergence time. However, they found their approach hard to scale, and it is not easily generalized.

8.4 Illustration of Tradeoff Analysis

We illustrate how using Condor's metrics, and rapid specification and synthesis of alternate topologies, allows an architect to explore tradeoffs, in this case between throughput degradation and convergence delay (RIR) following a failure.

For this example, we use a small 3-stage Clos built from 32 x 10 GbE switches, supporting 3072 hosts at 3:1 oversubscription. To reduce wiring complexity, we co-locate spine and aggregation switch nodes as linecards across 4 physical chassis, using backplane wiring to connect linecards.

Each of the ToRs has 2 uplinks to each of the 4 chassis. We compare two options for striping these uplinks:

Diversity-optimized (DO): Each ToR uplink connects to as many chassis linecards as possible (i.e., 2) at each chassis. For (DO), the failure of a single ToR uplink triggers routing table updates across a combined total of 15 ToR and spine switch nodes (RIR=2, because the impact of a link-failure propagates to aggregation and spine switches).

Convergence-time optimized (CO): Connect each ToR's uplinks to just one linecard per chassis. For (CO), assuming ECMP, the failure of a single ToR uplink does not trigger any routing table updates (RIR=0). However, the failure of a single linecard would cause each connected ToR to lose 25% of its uplink bandwidth.

Is (CO)'s faster convergence time worth its lower throughput in the case of a linecard failure? To decide, an architect would need to evaluate the SLO-compliance metric for one or more traffic matrices.[13]

For our evaluation, we use a uniform all-to-all traffic matrix of ToR-to-ToR flows. We set MTBF=150,000hrs and MTTR=24hrs for ToRs, linecards, and physical links. For

[13]The relative weights of SLO compliance and routing convergence is a business judgment outside our scope.

SLO throughput =	100%		99%		95%	
SLO percentile =	99	99.5	99	99.5	99	99.5
Diversity-optimized	**0**	**85.2**	**0**	**85.2**	**0**	85.2
Convergence-optimized	100	100	0.4	94	**0**	**1.7**

Cell values = % of flows experiencing SLO violations (**lower** is better)

Table 2: Comparing options (DO) vs. (CO)

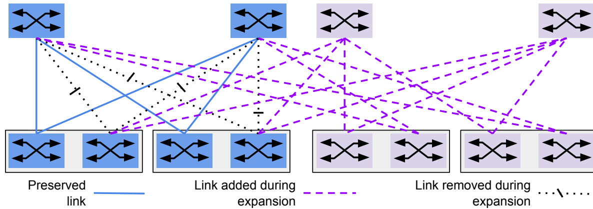

Figure 7: Changes to striping during fat-tree expansion (2 to 4 pods)

simplicity, we assume MTBF=∞ for other components.[14] We ran 10^6 iterations of MCMC for each striping option, which took 45 minutes on 200 non-dedicated cores using non-optimized Python code. It would be much faster in C++. (10^6 iterations is probably enough for high confidence, but we have not yet implemented a confidence metric.) For each option, over all iterations, we count how many flows are SLO compliant. An example SLO could be "achieve 95% of ideal throughput for 99.9% of the samples."

Table 2 shows that, for our simplified example, the fraction of ToR-to-ToR flows experiencing SLO violation varies widely. Option (DO) usually ties or beats (CO) — but not always. Condor thus exposes an architect to complex consequences of relatively simple design choices.

9. ENABLING ONLINE EXPANSIONS

Owners of large datacenters build them to handle years of future growth, initially deploying a small fraction of total server and network capacity because installing all network capacity at once is too expensive. Condor helps architects design topologies that support efficient, incremental expansion, and it also helps plan expansion operations.

While one could perform expansions offline by temporarily removing some or all of the workload, this approach results in costly downtime. We can perform expansions *online* if we can preserve the network's SLO during all phases of an expansion operation. Online expansion is tricky because, for many topologies, including trees and Jellyfish, expansion often requires moving (and thus draining) existing links. For instance, adding a new pod to a tree also requires adding one or more spines (overbuilding spines is too costly), which in turn requires redistributing links so that each pod has the same number of connections to each spine (see Figure 7).

The naïve solution of re-wiring at small increments (e.g., port-by-port) might preserve SLOs but is not cost-effective, leaves the network in a vulnerable state for too long, and increases operational complexity. Instead, architects must decompose an expansion into multiple stages, during which a set of ports is drained, re-wired, and un-drained. These stages must be designed so as to maintain the network's SLO during and after each stage, in the presence of expected failure rates. Condor helps architects design these expansion stages.

[14]The MTBF/MTTR values are illustrative but plausible. They lead to network-wide rates of a link failure/~73hrs and a switch failure/~669hrs, since links outnumber switches by 2048:224.

9.1 Expansions: Practical Considerations

Patch Panels: Architects can use patch panels to reduce the operational complexity of expansions by allowing all re-wiring to happen in one place. For example, one can pre-install cabling from all of a fat-tree's aggregation and spine switches to the back of the patch panels, and then make or modify inter-switch connections via the front of the panels.[15] (Some topologies, e.g., Jellyfish [33], do not support expansion via patch panels.) Condor's TDL and synthesizer can support patch panels and constraints on their wiring (§9.2).

For operational stability (humans working on patch panels can accidentally dislodge or unplug nearby cables), we prefer to design an expansion stage to drain an entire patch panel (or rack of panels), re-wire those ports, test the links, and then un-drain before moving to the next stage.

Constraints on Increments: During the lifetime of a tree topology that evolves by cost-effective expansions, it might be impossible to ensure that each aggregation switch has the same number of links to each spine. For example, in a five-pod tree topology, some pod-spine pairs would have p links, and some would have $p + 1$, for some p. Using ECMP on such unbalanced networks would lead to packet loss at spine switches, which forces operators to drain otherwise-working links to restore balance. An architect must decide, during network design, whether to adopt something other than ECMP (such as WCMP [44]), tolerate these link drains, or accept larger (and costlier) expansion increments. Condor helps architects evaluate the potential impact of unbalanced connectivity, to ensure that expansion increments maintain SLO compliance.

Other constraints besides imbalance apply to choosing expansion increments for other topologies. For instance, DCell [16] and BCube [15] become less fault-tolerant at certain increments. Condor supports ways to identify and avoid these issues, which for space reasons we do not discuss.

9.2 Describing Expansions with TDL

We designed TDL and the synthesizer to ensure that an architect does *not* need to write separate files for each increment of expansion. Instead, an architect describes an incrementally-expandable topology as it would exist following the *final* increment, including building blocks, relationships, connectivity pairs, and constraints. For instance, an architect could design a tree topology with an upper limit of 64 spine switches and 64 pods, with each pod containing 32 aggregation and 32 ToR switches, and then designate the pod as the unit of expansion. The architect can declare, in TDL, the rate at which the topology will expand ("2 pods"). Then, during synthesis of each increment of 2 pods, only

[15]To reduce complexity, an operator may disallow connections between front-side ports on two different panels.

the minimum number of spines required to meet the defined connectivity constraints will be instantiated.

We add a new building block class (*TDLCouplerBlock*) to support descriptions of patch panel ports and other passive connectors. In addition, we extend TDL's connectivity-pairs concept to support describing connectivity via patch panels and other passive devices (which are useful for purposes besides expansion [39]). First, an architect defines connectivity pairs that generate the connectivity from *endpoints* (such as spine and aggregation switches) to patch panels, expressing these pairs using building blocks and associating them with constraints, as in §5. The architect then adds an additional connectivity pair for each set of building blocks to be connected via the patch panels, referring to the patch-panel building block via an extra attribute. By defining constraints on this connectivity pair, the architect can constrain the number of connections between endpoints across *all* intermediary patch panels and/or across each individual patch panel.[16] The architect may also include constraints on the physical patch connections (e.g., that patch panel ports must be wired in pairs, to reduce the number of individual cables).

Late-bound constraints (§5.2.2) enable TDL to capture that the bounds on the number of pair-wise connections between spine and aggregation switches will vary as the network expands.

While we focus here on expanding tree topologies, the TDL and synthesizer can expand other non-random topologies.

9.3 Synthesis for Expansions

Synthesizing connectivity between patch-panel ports presents a challenge, due to the number of candidate connections. Recall (from §6.2) that this number, and thus the number of solver decision variables, is the size of the Cartesian product of the sets of components eligible for connection. If we constrain our use of patch panels to prevent patches that cross between panels, and we divide the ports on any panel equally between two types of components (e.g., aggregation and spine switches), and we use a decision variable for each port-pair, then, for N-port panels, we would need $(N/2)^2$ decision variables. A network using 192 256-port panels needs 3.14×10^6 variables, requiring lots of RAM and CPU.

To avoid this overhead, our synthesizer uses one decision variable for each port connecting to one of the two component types, rather than one per pair of ports, reducing the number per panel from $(N/2)^2$ to $N/2$. Assume we assign decision variables to ports connecting to aggregation switches. The synthesizer then assigns a unique index to each instance of the other component type (spine switches, in this case). It then sets the domain of the decision variables to include the range of this index, as well as 0 (indicating a port not used for patching). (Here, tiebreakers also allow architects to apply a rank-ordering to the set of *values* in each decision variable's domain, as the previously-described "greedy" heuristic (§6.3) no longer applies.) To ensure an injective mapping, the synthesizer adds a constraint: for each panel P and each spine switch S_i, the number of integer variables set to i is no more than the number of links between S_i and P.

[16] Constraints can also be defined on predecessors (racks).

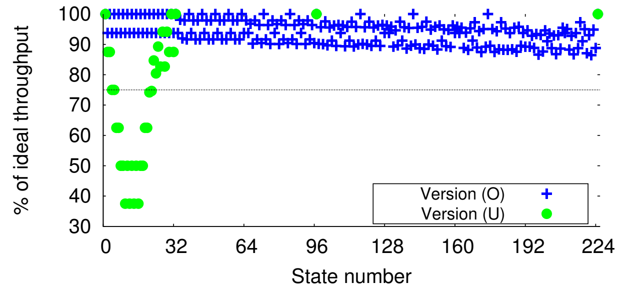

Figure 8: Throughput vs. state number

9.4 Does Condor Help with Expansions?

Does Condor help architects design expandable networks and expansions? We argue that it can, based on experiments with an example topology. We define an expandable network as one that meets its SLOs at all points in its *lifecycle*, including initial and expanded increments, and during expansions.

For this example, we used TDL to model a four-stage fat-tree using 256-port spine switch blocks and 512-port stage-3 aggregation switch blocks (these blocks are composed of smaller switches). Each pod is composed of ToRs, along with stage-2 and stage-3 aggregation switches. We use 256-port patch panels, grouped into racks containing 8 panels each, to support the reconfiguration of the striping between the stage-3 aggregation switches and spine switches during expansions. Our design oversubscribes ToR uplinks by 3:1, and the maximal network supports 49,152 end hosts, in 16 pods of 3072 hosts, connected by 64 spine switch blocks. The smallest possible increment of expansion is two pods.

Using TDL, we defined two versions of this network:

Unoptimized (U): includes constraints that establish balanced, symmetrical connectivity between all components in the maximal configuration. It does not include any constraints that specifically attempt to provide load balance for smaller configurations or during expansion stages.

Optimized for lifecycle throughput (O): Version U, plus a tiebreaker that prefers connecting to spine switches that have fewer connections, a heuristic hypothesized to ease expansion.

We expanded in units of 2 pods. For each expansion (2 to 16 pods), we synthesized a model using both versions. We assumed that operators split each expansion into 16 stages, each re-wiring 1 patch-panel rack. The network can be in $(7*16*2)+1=225$ states, counting the drained and undrained states for each of the 16 stages for each of the 8 increments.

We evaluated each network, at each state in its lifecycle, using our approximate bandwidth metric (§8.1), and configured it to divide flows at each junction as ECMP would.

We simplified the process of analyzing each stage of expansion by not performing "compensating drains" (draining additional links), a process used to maintain the "equality assumption" that ECMP depends on. (ECMP assumes

that all paths to the destination have equal bandwidth capacity [44]). Intermediate phases during an expansion operation can violate this assumption as drained links and rewiring create imbalance. For example, the first aggregation switch in pods 1 and 2 ($agg_{1,p1}$, and $agg_{1,p2}$ respectively) may both connect to spine switches $spine_1$ and $spine_2$. By default, $agg_{1,p1}$ and $agg_{1,p2}$ may both have 20 Gbps capacity to both spines, but during an expansion $agg_{1,p2}$ may only have 10 Gbps of capacity to $spine_2$ because of a drained link. With ECMP, $agg_{1,p1}$ will send traffic to $agg_{1,p2}$ through both spines equally, potentially resulting in packet loss at $spine_2$. To avoid this, operators can drain a link (referred to as a "compensating drain") between $agg_{1,p1}$ and $spine_2$ to equalize the capacity between each aggregation switch and $spine_2$. This simplification in modeling introduces some error, but (we believe) not consistently in one direction.

Figure 8 plots the percent of ideal throughput achieved (using the approximate bandwidth metric) vs. state number. An operator typically wants to maintain a target fraction of ideal throughput, such as the 75% line on the graph. Figure 8 shows that version **O**'s capacity dips during some expansion stages but never below 86%, so **O** is "expandable" with respect to the 75% target. We synthesized all 225 states in about 20 min (and this is easily parallelized). Without Condor, it would be much harder to validate **O**'s expandability.

Figure 8 shows two problems for version **U**. First, between 2 and 4 pods, we cannot perform expansion without dropping to 37.5% of capacity (well below the target). Without Condor's help, this would have been hard to spot.

Second, the missing tiebreaker in **U** caused the solver to backtrack for most other configurations; it cannot efficiently find a solution to the $(p, p+1)$ problem described in §9.1 before our 10-hour timeout. (It did solve the 8-pod and 16-pod states.) We know, from operational experience, that such solutions exist, but we do not know how to construct a tiebreaker that would allow efficient synthesis of **U** (as opposed to simply generating **O**). Existing research discusses heuristics for avoiding backtracking in CSP (e.g., [6, 12]), but we have not yet figured out how best to apply such techniques.

9.5 Prior Work on Expansions

Jellyfish [33] was designed to support incremental expansion, but the authors did not discuss the operational complexity of re-cabling large-scale networks, and it may be hard to design online expansions for large Jellyfish networks.

DCell [16] and BCube [15] support expansions, subject to constraints described in those papers. Condor gives architects a tool to quickly find expansions that meet those constraints.

Curtis *et. al* described algorithms for expanding Clos networks [9] and unstructured topologies [8]. Neither paper discussed how to optimize a design for future expansion or how to support online expansions. Liu *et. al* described zUpdate [24], a technique for coordinating the control and data planes to support non-disruptive online updates, including topology changes, but did not discuss the problem of topology design for expansions. Solnushkin [34], in presenting his algorithm for automated design of 2-layer fat-trees, discusses how to reserve switch ports for future expansion, but does not consider on-line expansion or more general topologies.

10. CONCLUSIONS

Designing a datacenter network requires balancing a range of goals and constraints, but no existing tool allows a network architect to easily explore tradeoffs across candidate topologies. In this paper, we presented Condor, our approach to supporting rapid exploration of the design space. Condor facilitates straightforward specification and exploration of designs with different tradeoffs, compared across a range of metrics. Condor's declarative, constraint-based Topology Description Language enables concise and easily modifiable descriptions of datacenter networks that Condor transforms into constraint-satisfaction problems to support rapid synthesis. We demonstrated that combining the power of TDL with Condor's analysis suite leads to new insights and helps architects navigate complex tradeoffs. Finally, we showed how Condor can support efficient expansions of traffic-carrying networks while preserving Service-Level Objectives. Overall, we believe Condor supports a much faster design cycle.

Acknowledgements

We thank Google colleagues for their insights and feedback, including Abdul Kabbani, Arjun Singh, Blaz Zupan, Chi-Yao Hong, David Zats, Eiichi Tanda, Fabien Viger, Guillermo Maturana, Joon Ong, Junlan Zhou, Leon Poutievski, Moray Mclaren, Google's Operations Research Team, Roy Alcala, Rui Wang, Vijoy Pandey, Wendy Zhao, and, especially, Matt Beaumont-Gay and KK Yap. We also thank Christopher Hodsdon, Tom Anderson and Vincent Liu, our shepherd Aditya Akella, and the SIGCOMM reviewers.

11. REFERENCES

[1] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. HyperX: Topology, Routing, and Packaging of Efficient Large-scale Networks. In *SC*, page 41, 2009.

[2] A. Akella, T. Benson, B. Chandrasekaran, C. Huang, B. Maggs, and D. Maltz. A Universal Approach to Data Center Network Design. In *ICDCN*, 2015.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, pages 63–74, 2008.

[4] A. Andreyev. Introducing data center fabric, the next-generation Facebook data center network. http://bit.ly/1zq5nsF, 2014.

[5] J. Arjona Aroca and A. Fernandez Anta. Bisection (Band)Width of Product Networks with Application to Data Centers. *IEEE TPDS*, 25(3):570–580, March 2014.

[6] C. Bessiére, A. Chmeiss, and L. Saïs. Neighborhood-Based Variable Ordering Heuristics for the Constraint Satisfaction Problem. In T. Walsh, editor, *Principles and Practice of Constraint Programming – CP 2001*, volume 2239 of *Lecture Notes in Computer Science*, pages 565–569. Springer Berlin Heidelberg, 2001.

[7] Cisco Systems. Cisco's Massively Scalable Data Center. http://bit.ly/1relWo8.

[8] A. R. Curtis, T. Carpenter, M. Elsheikh, A. López-Ortiz, and S. Keshav. REWIRE: An Optimization-based Framework for Unstructured Data Center Network Design. In *INFOCOM*, pages 1116–1124. IEEE, 2012.

[9] A. R. Curtis, S. Keshav, and A. Lopez-Ortiz. LEGUP: Using Heterogeneity to Reduce the Cost of Data Center Network Upgrades. In *CoNEXT*, pages 14:1–14:12, 2010.

[10] B. Dieter and H. Dietz. A Web-Based Tool for Optimized Cluster Design. http://bit.ly/1fyovAl, 2007.

[11] W. R. Dieter and H. G. Dietz. Automatic Exploration and Characterization of the Cluster Design Space. Tech. Rep. TR-ECE-2005-04-25-01, ECE Dept, U. Kentucky, 2005.

[12] D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *IJCAI*, pages 572–578, 1995.

[13] P. Gill, N. Jain, and N. Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *SIGCOMM*, pages 350–361, 2011.

[14] Google, Inc. or-tools: the Google Operations Research Suite. https://code.google.com/p/or-tools/.

[15] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *SIGCOMM*, pages 63–74, 2009.

[16] 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 *SIGCOMM*, pages 75–86, 2008.

[17] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring Network Structure, Dynamics, and Function using NetworkX. In *SciPy*, pages 11–15, Aug. 2008.

[18] H. Hanani. The existence and construction of balanced incomplete block designs. *The Annals of Mathematical Statistics*, pages 361–386, 1961.

[19] H. Hanani. Balanced incomplete block designs and related designs. *Discrete Mathematics*, 11(3), 1975.

[20] S. A. Jyothi, A. Singla, B. Godfrey, and A. Kolla. Measuring and Understanding Throughput of Network Topologies. *arXiv preprint 1402.2531*, 2014.

[21] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In *IMC*, pages 202–208, 2009.

[22] J. Kim, W. J. Dally, and D. Abts. Flattened Butterfly: A Cost-efficient Topology for High-radix Networks. In *ISCA*, pages 126–137, 2007.

[23] M. Li, D. Subhraveti, A. R. Butt, A. Khasymski, and P. Sarkar. CAM: A Topology Aware Minimum Cost Flow Based Resource Manager for MapReduce Applications in the Cloud. In *HPDC*, pages 211–222. ACM, 2012.

[24] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *SIGCOMM*, pages 411–422, 2013.

[25] V. Liu, D. Halperin, A. Krishnamurthy, and T. E. Anderson. F10: A Fault-Tolerant Engineered Network. In *NSDI*, pages 399–412, 2013.

[26] Y. J. Liu, P. X. Gao, B. Wong, and S. Keshav. Quartz: A New Design Element for Low-Latency DCNs. In *SIGCOMM*, pages 283–294, 2014.

[27] B. Mandal. Linear integer programming approach to construction of balanced incomplete block designs. *Communications in Statistics-Simulation and Computation*, 44(6):1405–1411, 2015.

[28] J. Mudigonda, P. Yalagandula, and J. C. Mogul. Taming the Flying Cable Monster: A Topology Design and Optimization Framework for Data-Center Networks. In *USENIX Annual Technical Conference*, 2011.

[29] S. R. Öhring, M. Ibel, S. K. Das, and M. J. Kumar. On Generalized Fat Trees. In *Parallel Processing Symposium*, pages 37–44. IEEE, 1995.

[30] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: Locality-aware Resource Allocation for MapReduce in a Cloud. In *SC*, pages 58:1–58:11. ACM, 2011.

[31] L. Popa, S. Ratnasamy, G. Iannaccone, A. Krishnamurthy, and I. Stoica. A Cost Comparison of Datacenter Network Architectures. In *CoNEXT*, pages 16:1–16:12, 2010.

[32] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hoelzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM*, 2015.

[33] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking Data Centers Randomly. In *NSDI*, pages 225–238, 2012.

[34] K. S. Solnushkin. Automated Design of Two-Layer Fat-Tree Networks. *arXiv preprint 1301.6179*, 2013.

[35] Z. Taylor and S. Ranganathan. *Designing High Availability Systems: DFSS and Classical Reliability Techniques with Practical Real Life Examples*. John Wiley & Sons, 2013.

[36] E. Tsang. *Foundations of Constraint Satisfaction*, volume 289. Academic Press, London, 1993.

[37] A. Varga et al. The OMNeT++ discrete event simulation system. In *ESM2001*, 2001.

[38] A. Varga and G. Pongor. Flexible topology description language for simulation programs. In *ESS97*, 1997.

[39] M. Walraed-Sullivan, J. Padhye, and D. A. Maltz. Theia: Simple and Cheap Networking for Ultra-Dense Data Centers. In *HotNets*, page 26. ACM, 2014.

[40] M. Walraed-Sullivan, A. Vahdat, and K. Marzullo. Aspen Trees: Balancing Data Center Fault Tolerance, Scalability and Cost. In *CoNEXT*, pages 85–96, 2013.

[41] X. Wen, K. Chen, Y. Chen, Y. Liu, Y. Xia, and C. Hu. VirtualKnotter: Online Virtual Machine Shuffling for Congestion Resolving in Virtualized Datacenter. In *ICDCS*, pages 12–21. IEEE, June 2012.

[42] F. Yates. Incomplete randomized blocks. *Annals of Eugenics*, 7(2):121–140, 1936.

[43] E. Zahavi, I. Keslassy, and A. Kolodny. Quasi Fat Trees for HPC Clouds and Their Fault-Resilient Closed-Form Routing. In *Hot Interconnects*, pages 41–48. IEEE, 2014.

[44] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *EuroSys*, page 5, 2014.