Catalyst: Unlocking the Power of Choice to Speed up Network Updates

Rohan Gandhi
Carnegie Mellon University, Microsoft

Ori Rottenstreich
Princeton University

Xin Jin
Johns Hopkins University

Abstract

Speeding up network updates is crucial to maintain high agility and to react quickly to network failures. In this paper, we present CATALYST—a new design to reduce the network update time. We observe that networks offer a power of choice, where there are many equally-good alternative paths that traffic flows can be assigned to, which is facilitated by redundancy in networks. CATALYST exploits this power of choice to assign flows to alternative paths to merge stages in the dependency graph (that captures the update plan), which in turn reduces the total update time. Furthermore, we observe that because of the prevalence of switch stragglers—switches that unexpectedly take longer time to update, simply assigning a flow to a single (shortest) path is not an optimal design as even a single switch straggler can substantially increase the update time. Thus, the second principle in CATALYST is to compute multiple paths for individual flows offline, among which one would be selected at runtime based on temporal switch conditions, in order to enable a fast update. Our evaluation using a load-balancer setting in a data center network shows that CATALYST effectively reduces the total update time by 1.14–2.15×.

CCS Concepts

• **Networks** → **Network management**;

Keywords

Network update, SDN, redundancy

ACM Reference Format:
Rohan Gandhi, Ori Rottenstreich, and Xin Jin. 2017. Catalyst: Unlocking the Power of Choice to Speed up Network Updates. In *CoNEXT '17: CoNEXT '17: The 13th International Conference on emerging Networking EXperiments and Technologies, December 12–15, 2017, Incheon, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3143361.3143397

1 Introduction

Centralized network control provides many benefits over traditional distributed control, from improving network utilization [8, 9], to enforcing policies [5] and reducing network energy usage [7]. The effectiveness of centralized control systems depends on how quickly they can adapt the data plane to network events like traffic, policy and topology changes. For example, B4 triggers data plane updates 540 times a day on average [9]. This puts a lot of burden on the control system to quickly compute a new network routing plan (called network state) and update the network accordingly.

Existing centralized control systems deal with network adaptations in two steps [8, 9, 13]. In the first step, they compute a *target state*, which assigns paths to flows to optimize an objective function (*e.g.,* minimizing average path length or maximizing total throughput) under topology constraints; in the second step, they compute a plan to update the network to the target state. It is important to update the network as fast as possible while providing consistency guarantees for the update period, *e.g.,* no loops, no blackholes and no link congestion [15, 17]. To avoid violating the consistency requirements, the network update often has to be performed in multiple stages rather a than single shot [8–10, 14]. As shown in production networks like B4, computing a target state is relatively fast, which takes between 0.1-1 second, while updating the network takes 3-5× more time in the median case [9].

Many solutions have been proposed to improve the speed of network updates using dynamic scheduling [10], verification tools [21], and through incremental updates [12]. However, all these network update solutions make one assumption — *the target state is given, and cannot be changed at runtime.* This assumption unnecessarily restricts network updates. If a target state is in itself onerous, then there is little an updater can do to reduce the update time. In contrast, we raise a question in this paper: *can we allow some flexibility in the selection of the target state to make the update faster?*

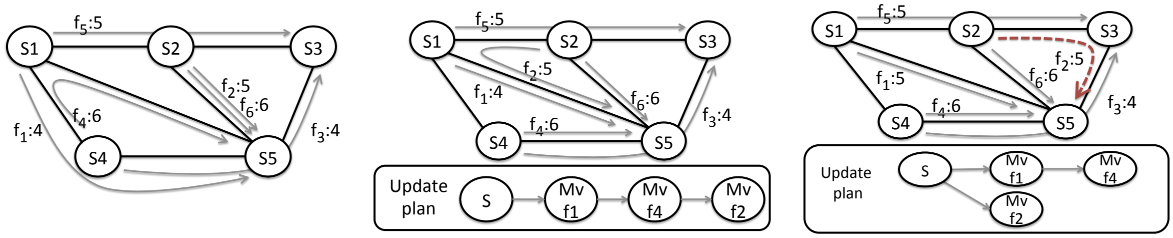
To that extent, we design CATALYST, a new approach to accelerate network updates. Instead of sticking with a single target state, CATALYST, first creates many choices of target states *offline* that are *close-to-optimal* but are *faster* to update. Then *online*, CATALYST dynamically selects one target state depending on the runtime conditions to speed up the update.

The first idea in CATALYST is to choose a target state that has *fewer stages in the dependency graph.* A dependency graph captures the dependencies between individual update operations, in order to meet the consistency requirements for a network update [10, 15]. It usually contains multiple stages, and the number of stages dominates the update time. We observe that because networks are typically provisioned with *redundancy*, there are multiple near-optimal paths to place a flow. For example, FatTree offers multiple paths of equal lengths between every pair of racks, or WAN topologies offer many equal-cost paths [9]. Carefully moving a flow to an alternative path may resolve a dependency between two flows and thus merge two stages. We leverage this flexibility to find a target state with fewer stages and speed up network updates.

However, because of switch stragglers, a new target state with fewer stages may not always be faster to update to if such target state encounters a straggler, whereas the original target state does not encounter a straggler. A switch straggler takes much longer time (e.g., 10×) to complete an update operation than other switches [9,

CoNEXT '17, December 12–15, 2017, Incheon, Republic of Korea

ACM ISBN 978-1-4503-5422-6/17/12…$15.00
https://doi.org/10.1145/3143361.3143397



(a) Current state (b) Optimal target state (c) Faster update plan

Figure 1: Initial, target network states with network update plans. $f_x : t$ denotes x-th flow requires t units of bandwidth. The link capacity is 10 units. Note that link S2-S5 cannot support the traffic in (a). (c) shows faster update plan when f_2 is routed differently (through S2-S3-S5 – red line). S in update plan denotes "start".

10]. An update involving a switch straggler can take a long time, even if it has fewer stages. The challenge is that a switch straggler can be temporary and thus is hard to predict beforehand.

To approach this uncertainty, our second idea in CATALYST is to compute *multiple* alternative paths for a flow offline, and then choose one of them online depending on the runtime conditions. While choosing the multiple paths (per flow), we want to reduce the number of switches common across the paths to limit the impact of straggler switches. We show that this problem can be formulated as a *max-cover* [2] problem, which is NP-hard but exists many approximation algorithms. To choose one path online among multiple alternative paths, we present *reactive* and *speculative* approaches that try different paths in series and parallel respectively (§4). We show the trade-off among these approaches in terms of update time and resource reservation overhead. We choose the speculative approach for its shorter update time.

In summary, we introduce the idea of using single and multiple alternate paths to speed up network updates: (i) we show how to compute a single alternative path for individual flows at scale and prove that merging the stages (in dependency graph) *is always expected* to reduce the total update time; (ii) we show how to compute multiple alternate paths using variants of well-known max-cover and bin-packing problems; (iii) we present a trade-off in update time and efficiency in the speculative and reactive approaches in choosing a path during run-time; (iv) our evaluation using a load-balancer setting in a data center network shows that CATALYST speeds up the network updates by 1.14-2.15×.

2 Catalyst Overview

In this section, we first briefly present background on existing network update schedulers, and then make a case for using alternative paths to improve network update speed.

2.1 Background on Network Updates

The network state is updated frequently to adapt to a variety of dynamics including changing traffic volumes, failures and policies. The network state includes the routes to the individual flows. Existing controllers [8, 9, 11, 13] first calculate the target state (target paths for individual flows) that maximizes an objective function while satisfying some policies. For example, B4 and SWAN maximize the WAN utilization, while ensuring the policies, such as no traffic blackholes or max-min fairness. In the second step, the network is updated from its current state to the target state while satisfying *consistency properties*, *e.g.,* no loops or congestion during updates.

The goal of the update step is to update the network safely and quickly. Once the initial and a target states are given, the fastest way to update (add/remove/edit paths) is to apply all update operations in one shot. However, such approach can violate consistency requirements (*e.g.,* transient congestion, loops and traffic blackholes) [17]. The consistency requirements impose *dependencies* on the order on which network updates can be applied. For example, for congestion freedom, a new flow cannot be added before an old flow is removed. Many solutions have been proposed to speed up network updates to meet consistency properties [10, 12, 21]. Based on the dependencies, these approaches construct a *dependency graph*, where each stage consists of a set of updates that can be applied in parallel, whereas updates in one stage cannot be started until all operations in the prior stages have completed. As a result, the total update time depends on the number of stages in the dependency graph. As reported in B4, the network update takes 3-5x more time compared to the target state computation for Traffic Engineering (the computation takes 0.3 seconds in the median case). Similar results are also reported in SWAN [8] and Dionysus [10].

2.2 The Case for Alternative Paths

Existing controllers decouple target state computation from network updates [8, 9, 11, 13]. They make one assumption — *the target state given to the network updater is fixed and cannot be changed.* Under this assumption, if the target state is computed such that the update plan is onerous, then there is little the updater can do to speed up the network update. Target state computation may in fact end up with a state with high update latency because of two reasons. (i) The goal of the target state computation is to optimize an objective function (*e.g.,* maximize WAN utilization, minimize average latency), but not about optimizing the update latency. As shown in Fig. 1, flow f_2 has two paths, S2-S1-S5 and S2-S3-S5 that are equally optimal (same number of hops), and path with longer update time (S2-S1-S5) can be chosen. (ii) Importantly, in many cases [8, 9], the number of stages that govern the update speed can only be calculated once the entire target state for *all flows* is known. Thus, the target state computation may not even account for the update speed during computation.

Our key observation is that production networks are usually provisioned with redundancy to accommodate failures. Such redundancy provides many alternative paths to place flows without sacrificing the optimality of the target state. For example, data center topologies like FatTree contain multiple equal-length paths between two racks which can be used to place flows. Similarly, wide area networks also have multiple paths between two data centers which provides flexibility for flow placement. Thus, the key idea in CATALYST is to *reassign flows to these alternate paths to merge the stages in the dependency graph*. For example, as shown in Fig. 1(c), if the target state is changed to the one where f_2 is assigned to path S2-S3-S5 (red line), the network update can be made faster (see update plan in Fig. 1(c)) while still providing same optimality. In §2.3, we prove that merging the stages is always expected to improve update time.

Moreover, if one alternative path can be found, then there are chances that more than one alternative paths can be found that meet the goals (*e.g.,* multiple equal-lengths paths in datacenter network). In such cases, a natural question is whether we need multiple alternative paths. As shown in Dionysus [10], switches *straggle*, *i.e.,* some switches occasionally take longer time to update the rules compared to other switches. The distribution of the update time exhibits a long tail, where the update time at 90^{th} percentile takes 10× of the median [10]. Thus, it may be possible that a single alternative target state that contains a few stages may turn out to be a slow one when installing updates at the runtime due to straggling switches. Thus, we argue that multiple target states should be computed offline, and the updater should dynamically choose one among them at the runtime to reduce the update time. This idea is synergistic with the principle of dynamic updates in Dionysus: by making more candidate target paths available for individual flows, it only increases the available paths to reach a target state faster.

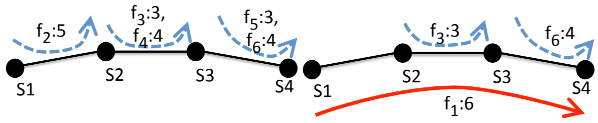
Also, compared to prior work, CATALYST is complimentary to FFC [13], which assigns traffic to multiple tunnels to reduce the impact of failures. CATALYST uses *existing* multiple paths to speed up the network updates. In contrast to ESPRES [16] that leverages inherent independence among the network updates with respect to the traffic, CATALYST *creates* more opportunities to exploit such independence. In contrast to the speculative task execution in big data analytic systems [3] or tail latency reduction [19, 20], CATALYST focuses on reducing damage from stragglers in network update.

2.3 Is Merging Stages Always Helpful?

The key idea in CATALYST to is to merge the stages in a dependency graph. Merging stages affects network update latency in opposite ways. It may seem natural to reduce number of stages to reduce network update latency. However, note that, when two stages are merged, the number of updates in the merged state also increases, which increases the probability of a straggler, which can potentially inflate the network update latency. Thus, an important question is whether such merging is always expected to improve the overall update latency. As we show in this section, the answer is always yes.

We use the following probabilistic model. Let p denote the probability of an update to straggle. The probabilities are independent for different updates. Assume two stages in which n_1 and n_2 are the number of updates. Let t and s denote the update time of a stage without and with a straggling switch, with $t << s$ [10]. With probability (w.p.) $(1-p)^{n_i}$ none of the updates in stage i straggle. Thus, the expected time for stage i is $E_i = (1-p)^{n_i} \cdot t + (1-(1-p)^{n_i}) \cdot s$. For the two stages without merging is $T = E_1 + E_2$.

For simplicity, assume there is no change in the number of updates when merging the stages. Thus, the time of the merged stage is t w.p. $(1-p)^{n_1+n_2}$ and the expected time is $T' = E' = (1-p)^{n_1+n_2} \cdot t + (1-(1-p)^{n_1+n_2}) \cdot s$.



(a) Current state (b) Target state

Figure 2: Part of the network. Flow f_1 is added in the target state, and flows f_2, f_4, f_5 are moved (destination not shown). f_x : t denotes that x-th flow requires t units of bandwidth.

To show that the expected time in the merged stage is always shorter, *i.e.,* $T' < T$, we denote:

$T = 2t + (1-(1-p)^{n_1}) \cdot (s-t) + (1-(1-p)^{n_2}) \cdot (s-t)$, or

$T = 2s - ((1-p)^{n_1} + (1-p)^{n_2}) \cdot (s-t)$, or

$$T = s - (1-p)^{n_1} \cdot (s-t) + s - (1-p)^{n_2} \cdot (s-t) \qquad (1)$$

Similarly,

$T' = t + (1-(1-p)^{n_1+n_2}) \cdot (s-t)$, or

$$T' = s - (1-p)^{n_1+n_2} \cdot (s-t) \qquad (2)$$

From (1) and (2), $T - T' = s - ((1-p)^{n_1} + (1-p)^{n_2} - (1-p)^{n_1+n_2}) \cdot (s-t)$. Note that, $(1-p)^{n_1} + (1-p)^{n_2} \leq 1 + (1-p)^{n_1+n_2}$ for any probability p.

Thus, $T - T' \geq s - 1 \cdot (s-t)$, or $T - T' > 0$. Thus, $T' < T$.

Discussion: The effectiveness of CATALYST is bound to the redundancy (spare capacity) in the network. Datacenters tend to have low link utilization [4, 6, 18] making more capacity available for network updates. However, other networks such as ISP or wide area networks [8, 9] may not have such sparse traffic. For example, B4 and SWAN drive towards 100% network utilization. However, to provide high availability even during failures, such networks either reserve bandwidth or use bandwidth reserved for the lowest priority traffic as a cushion during failures [8]. CATALYST can re-purpose such bandwidth to speed up network updates. Our evaluation (§5) shows that even 10% spare capacity can provide dramatic reduction in the network update latency.

Lastly, in CATALYST, network operators do not know the exact paths that will be installed apriori. We believe this as a minor inconvenience because after all CATALYST does not hide the installed paths from operators – just that CATALYST reports the paths after they are installed.

3 Single Alternative Path

In CATALYST, when moving the flows to the alternative paths to merge the stages in the dependency graph, two major questions are: (i) which flows to move to the alternative paths; (ii) how to choose the alternative paths. In this section, we limit the number of alternate paths (per flow) to 1. Naively, if there are p_f paths for flow f, then there are a total of $\prod_f p_f$ alternative paths for all flows. Scrolling through each combination would be impractical at scale. Using an ILP (not shown due to space limit) to reduce the length of the dependency graph is not scalable — it takes roughly 70 seconds to find alternate paths even at a small scale for hundreds of flows with thousands of nodes. Below, we detail the heuristics to reduce the stages of the dependency graph at scale.

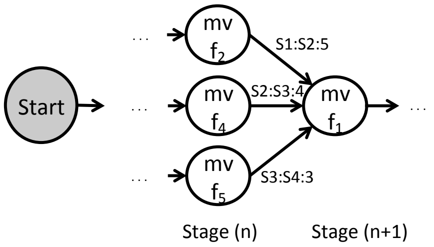


Figure 3: Dependency graph for Fig. 2(b). Note that moving flows f_2, f_4, f_5 may depend on moving other flows from previous stages (not shown). The resources freed are shown on the arrow. $Sx : Sy : z$ means "z" units are freed on link $Sx : Sy$.

3.1 Merging Stages in Dependency Graph

We observe that there exists a dependency between moving two flows because one flow is waiting for the *resources* that will be freed when another flow moves away. As shown in Fig. 3, flow f_1 is waiting on the link capacity resources that will be freed when flows f_2, f_4 and f_5 move away. Thus, we deduce the following two ways in merging a stage with a previous stage.

(1) **No resource dependency:** Move flows (f_1) from the stage to other paths to avoid dependency on the resources released by previous stage (*e.g.,* f_2, f_4, f_5), and eliminating that stage. By doing this, all flows from both the stages can be moved *concurrently*.

(2) **Resources available faster:** Move *other* flows (*e.g.,* f_3, f_6 instead of f_4, f_5) to other paths to create resources for f_1 faster.

In CATALYST, we first compute the dependency graph given the current and target network state. The dependency graph generation is similar to Dionysus [10], which we omit due to space constraints. We then traverse the stages in the dependency graph in the reverse direction, *i.e.,* start from the last stage and move up to the first stage while merging a given stage with its predecessor. This is because, moving flows from the successor node (the stage that depends on resources) to the alternate paths would potentially eliminate that stage, and may also eliminate predecessor stages from the dependency graph (its compliment is not true).

Once a stage is selected, all the flows in that stage must be assigned to alternative paths to merge that stage. We consider all the flows (in a stage) one-by-one for reassigning the paths. Finding the alternative paths at scale is challenging. For example, when one flow is assigned to a path (*e.g.,* flow f_1), it may have to displace some other flows (*e.g.,* flow f_4). These flows in-turn have to displace some other flows (*e.g.,* flow f_7 – not shown in Fig. 2(b)). Such displacements in chains create more dependencies and complicate finding paths for even a single flow, and is not tractable at scale.

Thus, in CATALYST, we assign a path (path-p) to a flow only if the path already has the resources without moving any other flows, *i.e.,* (i) all the links in path-p have enough capacity to handle traffic of that flow, (ii) all the switches have enough memory to store the rules. Further to quickly find the paths to the flows, we: (i) consider the paths (consisting of multiple links) in the decreasing order of the *bottleneck* capacity, (ii) In parallel, we also find the paths with high overlap with the *initial* path to reduce the number of updates. When shortlisting candidate alternate paths, we select paths that are equally optimal to the path in the initial target state. In this paper, we consider the distance between the end-points as the optimality value. Also, if a small degree of sub-optimality in the target state is allowed, it only amplifies the chances of finding the suitable alternative paths for the individual flows.

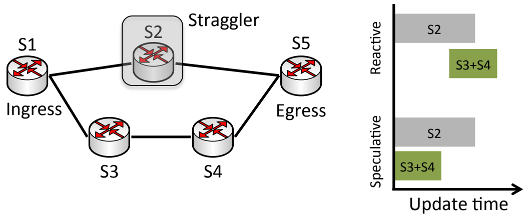


Figure 4: There are two paths from Ingress to Egress switches (S1-S2-S5, S1-S3-S4-S5). In *reactive*, paths are selected one by one, whereas in *speculative*, both paths are selected simultaneously. The corresponding update times are also shown.

In addition to the optimality check as above, we choose a path such that: (i) the movement of the flow can be merged with the previous stage, which is easily done by selecting the paths that do not require the resource that caused the dependency. For example, in Fig. 3, when merging stage (n+1), we find the alternate paths for flow f_1 that do not require resources on links S1-S2, S2-S3 and S3-S4; (ii) importantly, the number of updates in an alternate path does not exceed the updates in the original path, which ensures that we don't increase the probability of encountering a straggler when installing a path. We break the ties at random.

The second way to merge stages is to make resources available faster. To do so, we again only find paths that already have the resources to reduce the complexity. We leave further optimizations including moving other flows when assigning one flow for the future work. Also, this algorithm to choose paths can be easily parallelized as optimality of each path can be considered in parallel.

4 Multiple Alternative Paths

In this section, we explain the advantages of multiple paths for a flow. We describe how to select multiple candidate paths and then during update time how to select *one of them as an actual path.*

4.1 Managing Selected Paths

As shown in Dionysus [10] and B4 [9], switches occasionally straggle, and if a switch straggler is on an alternative path, then moving a flow to an alternative path would actually be slower. To handle such situations, we argue that multiple paths be computed offline, and one of them be chosen online dynamically based on the occurrence of straggling switches. To take advantage of multiple paths, we need to decide (i) how many alternative paths to choose for each flow, (ii) how to calculate alternative paths, and (iii) how to choose one of those paths at the runtime.

How to choose one from multiple alternative paths: There exists a trade-off between how many paths to compute and how to choose the paths. Imagine we have n paths for a flow, and the goal is to choose one from these paths. As switches straggle, there are two ways to choose the path. (i) *Reactive:* the updater selects one of the n paths, and starts installing that path. If it hits a switch straggler, it selects a new path and starts installing it, while simultaneously rolling back the previous partial update on the old path (Figure 4). The drawback of the reactive approach is that the updater has to wait until it detects a switch straggler, and then starts to update the network to a new alternative path. Both these actions can inflate the update latency. (ii) *Speculative:* the network updater selects two or more paths and *simultaneously* start making the update. The path that is installed first will be assigned to the flow. At the same time, the updater will roll back the update for other paths that are in progress. The speculative approach masks the impact of straggler switches as there are multiple alternative paths whose updates are in progress.

However, unlike the reactive approach, the speculative approach needs to pre-reserve the resources on all the candidate paths so that it can add the paths simultaneously, and thus suffers from high resource reservation overhead — if there are three alternate paths for a flow, then to simultaneously make the update for these three paths, it would potentially (worst-case) triple the memory and link consumption on corresponding switches and links.

In Catalyst, even though the speculative approach has higher requirements in terms of memory and link usage, we select this approach as it masks the slowdown caused by straggler switches. We select up to k paths (to conserve the resources), and start the update simultaneously. Once one of the k paths is successfully installed, we set that path at the ingress. Because we pre-reserve the resources, importantly, rolling back the remaining paths is simple as it only requires removing the rules, which can be done in background without adding any new dependencies and elongating update latency.

4.2 Computing Multiple Paths Offline

As we compute up to k paths (per flow) offline, the next question is how to compute these paths. There are two options. (i) Choose paths with *high overlap*, *i.e.,* multiple paths share common links to reduce the resource (switch, link) overhead. The common resources are to be reserved just once even when shared by multiple paths. (ii) Choose paths with *low overlap* to avoid straggler switches affect large fraction of the paths but has higher resource overhead.

In Catalyst, we choose the later option that uses the paths with only a small set of common switches as it has a smaller impact of straggler switches. In fact, we observe that this problem of selecting the paths with the least overlap is a variant of the *max-cover* problem [2], which aims to find the minimum set of paths (given a set of paths) whose union covers maximum nodes in the graph.

Computing *k* paths: In Catalyst, we compute the multiple alternative paths to the same flow inspired by the max-cover problem (NP-hard). Intuitively, we want to be resistant to a single straggler switch by having at least one path that does not include that switch. We refer to a node as a covered node if there exists an alternate path that contains that node. The details are the following. (i) The input to the algorithm includes the set of all paths for a given flow (§3). The output contains a subset of these paths. The number of paths is limited to k. (ii) We initially reset the set of covered nodes (Set $S = \emptyset$). At each stage, we find a path (set of nodes P_i) that maximizes the set of covered nodes (select P_i that maximize $|P_i \cup S|$) and then adjust the set of protected nodes ($S = S \cup P_i$). (iii) We continue until all k paths are selected or all nodes are exhausted, or no more paths are left due to limited resources.

Choosing *l* out of *k* paths: Note that when the above formulation computes up to k paths for the individual flows, it assumes that all the available paths in the network are available to the given flow and there are no further flows when computing paths. In reality, the network has many flows. Scrolling flows one-by-one and computing the paths is unfair as the flows selected in the beginning will have more resources available and thus more paths will be selected compared to the flows in the end. To avoid these problems, we further prune the number of paths for individual flows. The input to the algorithm includes all the computed paths for all individual flows by the max-cover algorithm (k). The output contains only a sub-set of the paths (l) with an objective to *maximize the minimum paths assigned per flow*. This is a variant of a multi-dimensional bin-packing algorithm with each dimension as a network resource (switch memory, link capacity), and the individual paths of the flows as the objects. We solve it using an ILP formulation shown in Fig. 5 using notations shown in Table 1. Constraint (1) denotes that the resource capacities are not violated when assigning the paths to the flows, while constraint-2 denotes that y is minimum number of paths across all flows.

Table 1: Notations used in the algorithm.

Notation	Explanation
Input	
F, P	Sets of flows and paths
C^r	Capacity of r-th resource
$U^r_{f,p}$	Resource utilization on r-th resource when f-th flow is assigned to p-th path
Output Variable (binary)	
$x_{f,p}$	set if f-th flow is assigned to p-th path
y	Min. number of paths for a flow

ILP Variable: $y, x_{f,p}$
Objective: Maximize y
Constraints:

Resource capacity: $$\forall r \in R, \sum_{p\in P, f\in F} x_{f,p} \cdot U^r_{f,p} \le C^r \tag{1}$$

Expressing y: $$\forall f \in F, y \le \sum_{p\in P} x_{f,p} \tag{2}$$

Figure 5: ILP formulation.

Discussion: The value of k depends on the redundancy in the network. A topology with high redundancy may contain larger pool of alternate paths, and larger values of k can be set. Larger value of k indicates more paths available to select alternative paths in the ILP. However, higher value of k also increases the time to find the paths using max-cover algorithm, and to solve ILP.

5 Evaluation

We implemented Catalyst using Python and C++, and used Cplex [1] to solve the ILP. We evaluate Catalyst using a load balancer scenario for datacenters (DCs) similar to [12]. The datacenter network has a FatTree topology which contains 1000 ToRs in 50 containers, and each link has a capacity of 10Gbps. We set the number of flows to 10K and available server replicas in load balancer to 100. The load balancer pre-computes the assignment between flows and replica servers, and adds the OpenFlow rules to all the source ToRs where the flows initiate to change the destination IP to the replica server assigned to that flow, and forwards the packets to the selected replica. The experiments show the results of changing one load balancer policy with another. The initial load balancer policy assigns a server replica to every flow uniformly at random. In the second load balancer policy, we fail 2 ToRs, and repeat the procedure for a different random assignment – again distributed among ToRs uniformly at random. Further, we set the limit on number of flows handled by the individual replica to $\left\lceil \frac{10K}{98} \right\rceil = 103$. Further, we set

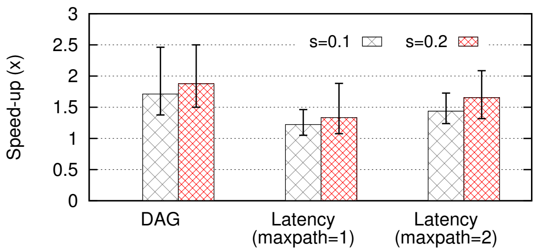


Figure 6: Improvements in CATALYST.

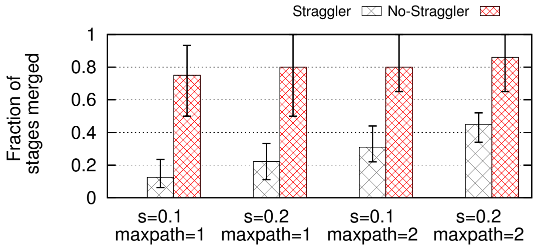


Figure 7: Straggler and non-straggler stages merged.

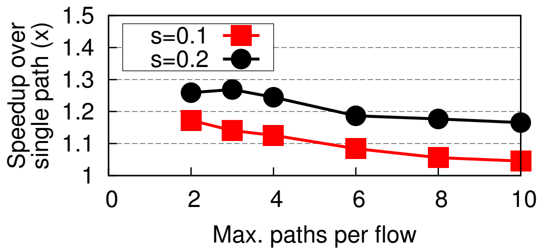


Figure 8: Impact of max. paths on the speedup.

aside *additional memory* (s) at the switches used during the network updates. We run experiments with additional memory set to 10% and 20% (s=0.1, 0.2). We do not consider s=0 as without additional memory, the network update may run into deadlocks [8].

We evaluate the impact of various design policies in CATALYST on the network update speed. We repeat each experiment for 100 times. We set the max. number of paths in the max-cover problem (k) to 10, and set the number of alternate paths chosen to 1 or 2 (l = 1,2). The computation time to find alternate paths takes 0.1-0.3 sec (median 224 msec). We calculate speed up for each of the 100 runs. Fig. 6 shows the speedup in median case while the error-bars denote the minimum and maximum speedup.

DAG stages (no straggler): First, we show the speedup in the DAG stages, *i.e.,* reduction of the DAG stages by assigning the flows to the alternate paths. The speedup is simply the ratio of the number of stages before and after merging the stages. Note that, the speedup in the DAG stages shows the speedup in network update when there are no stragglers as the update speed without stragglers only depends on the number of stages. CATALYST achieves impressive speedup of 1.7× (median) and 2.5× (peak) for $s = 0.1$, and similar speedup for $s = 0.2$ showing that CATALYST is highly effective in finding alternate paths to reduce the stages in the DAG.

Update latency (single alternate path): From the network updater prospective, the most important metric is the reduction in the total update time, which is shown in Fig. 6. First, we set the max. alternate paths (per flow) to 1, *i.e.,* at most 1 alternate path will be found for a given flow. As the update time of a stage depends on the max. update time of a flow in that stage, the total update time for S stages is $\sum_{s \in S} max(t_{f,s})$, where $t_{f,s}$ is update time for f-th flow in s-th stage. We model $t_{f,s}$ (including $t_{f,s}$ due to stragglers) using the distribution from [10]. The flows experiencing stragglers are selected at random. The speedup in the update latency is 1.05-1.88× (median = 1.22× and 1.32× for s = 0.1 and 0.2, respectively), which is surprisingly smaller compared to the improvements in the DAG stages. Figure 7 sheds more light on the loss in the speedup. Recall that the total update time is also dominated by the stages that observe stragglers. Figure 7 shows that although CATALYST reduces number of stages by 71% (for s=0.1), the fraction of stages that observed a straggler (denoted using f_1, and shown in black in Fig. 7) was reduced by only 12.5%. The key reason for the smaller f_1 is because there are just many flows in those stages and not all the flows could be assigned to the alternate paths. As we increase the extra capacity in the network to 20% (s=0.2), f_1 is also increased to 23% and this improved the overall speedup by 33% (median).

Update latency (multiple alternate paths): Next, we measure the impact of using multiple alternate paths for individual flows. Unlike previous experiment, in this experiment, as there can be multiple paths for individual flows, the $t_{f,s} = min(t_{f,s,p})$, where $t_{f,s,p}$ is the update latency for f-th flow in s-th stage for its p-path, which we again model using the distribution shown in [10]. In Figure 6, we show the results for maximum alternate path set to 2. As shown, CATALYST substantially improves the network update speed by 1.14-2.15× (median = 1.43× for s=0.1, 1.65× for s=0.2). If each flow in a stage has $a > 1$ paths, and if the probability of a straggler is q, then the probability that all the paths for that flow encounter a straggler is q^a, which is significantly smaller than the case of having $a = 1$. This reduction in the probability is directly reflected in the increase in the speedup achieved – compared to previous experiment (maxflow=1), we found higher values for f_1 = 31% and 43% for s=0.1 and s=0.2 respectively.

In the last experiment, we measure the sensitivity of the improvements in CATALYST by varying the maximum alternate path (M) between 2 to 10. As shown in Fig. 8, as M increases, the improvement in CATALYST gradually reduces, and almost reaches to 1× at $M = 10$. The key reason for the reduction is that after a certain value of M, the extra paths in the merged stages have no impact as there are already enough paths to substantially reduce the chances of a straggler, but that reduces the number of such extra paths for the flows in the un-merged stages, which increases the chances of a straggler, and thus increase in the update latency.

6 Conclusion

We argue that the network update speed can be substantially improved using a power of choice between multiple equally good paths. We present CATALYST, which uses the power of choice to shrink the dependency graph of the network updates and speedup the network updates. Further, we argue that because network switches occasionally straggle, multiple alternative paths be computed offline, and one of those paths be chosen online to reduce the impact of the straggling switches. In CATALYST, we find the alternate paths using a combination of max-cover and bin-packing algorithms. Our evaluation in load balancer settings shows that CATALYST improves the update time by 1.14-2.15×.

References

[1] IBM CPLEX LP Solver. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[2] Max cover problem. https://en.wikipedia.org/wiki/Maximum_coverage_problem.

[3] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Why Let Resources Idle? Aggressive Cloning of Jobs with Dolly. In *USENIX HotCloud* 2012.

[4] Theophilus Benson, Aditya Akella, and David Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC* 2010.

[5] Martín Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, and Scott Shenker. Rethinking Enterprise Network Control. *IEEE/ACM Trans. Netw.* 17, 4 (2009), 1270–1283.

[6] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *SIGCOMM* 2016.

[7] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree: Saving Energy in Data Center Networks. In *USENIX NSDI* 2010.

[8] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-driven WAN. In *ACM SIGCOMM* 2013.

[9] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. In *ACM SIGCOMM* 2013.

[10] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic Scheduling of Network Updates. In *ACM SIGCOMM* 2014.

[11] Nanxi Kang, Monia Ghobadi, John Reumann, Alexander Shraer, and Jennifer Rexford. Efficient traffic splitting on commodity switches. In *ACM CoNEXT* 2015.

[12] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental Consistent Updates. In *ACM HotSDN* 2013.

[13] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic Engineering with Forward Fault Correction. In *ACM SIGCOMM* 2014.

[14] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *SIGCOMM 2013*.

[15] Ratul Mahajan and Roger Wattenhofer. On Consistent Updates in Software Defined Networks. In *ACM HotNets* 2013.

[16] Peter Pereíni, Maciej Kuzniar, Marco Canini, and Dejan Kostić. ESPRES: transparent SDN update scheduling. In *HotSDN* 2014.

[17] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent Updates for Software-defined Networks: Change You Can Believe in!. In *ACM HotNets* 2011.

[18] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network’s (datacenter) network. In *SIGCOMM* 2015.

[19] Ashish Vulimiri, Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low latency via redundancy. In *ACM CoNEXT* 2013.

[20] Ashish Vulimiri, Oliver Michel, Brighten Godfrey, and Scott Shenker. More is less: reducing latency via redundancy. In *ACM HotNets* 2012.

[21] Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and Brighten Godfrey. Enforcing Customizable Consistency Properties in Software-defined Networks. In *USENIX NSDI* 2015.