Resilient Advertisements

Paper #559, 12 pages body, 19 pages total

Abstract

Abstraction

1 Introduction

Cloud and content providers (hereafter Service Providers) enable Internet applications used daily by billions of users. The applications have increasingly stringent performance and reliability demands, as the Internet is increasingly used for mission-critical applications (*e.g.,* enterprise services [**?**]) and as performance requirements become tighter (*e.g.,* virtual reality requires **TBD: X** ms latency and **TBD: Y** ms jitter [**?**]). To meet these stricter requirements with the same fundamental Internet protocols, Service Providers have deployed interconnected sites and connected with thousands of networks in hundreds of locations, offering rich connectivity and distributed capacity.

The deployment's physical resources (*i.e.,* peering links, servers) are provisioned to accommodate demands from (ingress) and to (egress) user networks. Service Providers balance egress demands across links to minimize congestion and latency [14, 17, 5]. However, it is difficult to control exactly how much traffic reaches each site and how much *ingresses* each link into the Service Provider's network since (a) ingress traffic can only be controlled indirectly, by directing users to different *prefixes*, and (b) it is difficult to advertise prefixes in a way that balances load in dynamic network conditions.

Given this lack of control, resource overload occurs due to link failures [10], DDoS attacks [**? ? ? ?**], flash crowds [**?**], and route changes that cause significant traffic shifts [**? ? ? ? ?** 10]. This overload leads to degraded service for user traffic [**? ? ?**]. Service Providers may respond to resource congestion by (a) overprovisioning resources [**? ?**] or (b) draining congested links and moving some of that traffic to other resources with free capacity [10]. We demonstrate in Section 2.3 that overprovisioning resources can require overprovisioning rates as 70% more , significantly increasing expenses.

Recent work drains traffic from overloaded links/sites by *withdrawing* prefix announcements that are also advertised via other (healthy) links/sites [**?** 10**?**], so that traffic destined to those prefixes is forced to arrive on other links/sites after BGP converges (within tens of seconds [**?**]). However, this solution to overload is post-hoc, unpredictable, and ineffective. Other solutions use traffic engineering systems in user networks to optimize latency and dynamically shift traffic [**?**], but do not provide guarantees that good backup options exist (and sometimes do not — see Figure 2).

To solve these problems, we propose a system — SCULPTOR— that proactively advertises multiple prefixes to peers to expose diverse routing options, then assigns user traffic to paths towards these prefixes to optimize latency and minimize overload during both steady-state and failure. That is, SCULPTOR ensures that advertisements are resilient to changes in traffic distributions and failed sites/peering links. Our key insight is to frame this problem as finding a way of advertising reachability that maximizes latency subject to capacity constraints both with and without failures, as congestion can be viewed as a partial deployment failure. Our solution leverages the rich connectivity of Service Providers, balancing traffic over backup paths through many sites and links as conditions change. We solve different research challenges than in the intradomain routing fault-tolerant-networking literature [**? ? ? ?**] since, unlike in that domain, we can neither control nor predict paths perfectly from users to Service Providers (interdomain routes are determined by BGP and thus by intermediate organizations). Instead, we build a model of how users are routed to Service Providers, and iteratively refine this model using measurements while using the model to optimize advertisement strategies using gradient descent.

We make two contributions. First, we present an optimization framework that can be used to minimize performance metrics (*e.g.,* maximum link utilization, latency, latency under single link failures) over different sets of prefix advertisements. Part of our framework is a model for predicting performance in unknown scenarios—we compute probability distributions of performance metrics in advertisement configurations we have not measured, which is important as changing routing configurations and then issuing measurements is slow. This model enables SCULPTOR to efficiently optimize over a large space without measuring every possible configuration. We demonstrate that this model quickly learns how to accurately predict performance metrics using few measurements compared to alternate approaches. We then optimize these (modeled) performance metrics using gradient descent which performs particularly well in our setting due to the high dimensionality of the problem and high degree of parallelism gradient descent admits.

Second, we prototype and evaluate our framework in a system, SCULPTOR, at Internet scale using the PEERING testbed [13], which is now deployed at 32 Vultr cloud locations [16]. Vultr is a global public cloud that allows us to issue BGP advertisements via more than 10,000 peerings. We demonstrate that SCULPTOR effectively computes prefix advertisements that give users low latency both during normal operation and variable network conditions.

SCULPTOR is able to reduce median latency under single link failures by **TBD: X** ms compared to the state of the art. We model SCULPTOR's impact under a variety of ingress link capacities (*i.e.,*, having more available capacity means there are fewer problems to solve, but comes with more cost). We demonstrate that SCULPTOR reduces necessary ingress capacity by **TBD: Y** times compared to the state of the art while retaining **TBD: Z** percent of the benefit. Hence, SCULPTOR serves both as an optimization tool for current deployments and as a network-planning tool—informing us where capacity is and is not needed.

2 Motivation and Key Challenges

2.1 Setting

Service Providers offer their services from tens or hundreds of geo-distributed sites. Each site offers identical service so can serve any user, but users want to reach a low-latency site for performance. Sites consist of sets of servers which have an aggregate capacity. Service Providers also connect to other networks (often thousands) at sites via dedicated links or shared IXP fabrics. Each such link also has a capacity. When utilization of a site or link (collectively, a resource) nears/exceeds 100%, performance suffers, so Service Providers strive to keep utilization reasonably low. Resources can also fail completely due to, for example, physical failure and misconfiguration.

Ensuring users can reliably route over healthy paths to Service Providers even during partial system failure is increasingly important as shown by the considerable attention that related problems have seen in the news and research community. Microsoft recently stressed the importance of resolving congestion on ingress links [10], developed a system to identify performance problems on paths [**?**], and is considering deployment deep within user networks to further enhance ingress routing performance and reliability [**?**]. Google similarly developed a system to identify performance problems on paths [8]. Peering disputes regularly make their way into the news, and these disputes often lead to long-standing congestion on interdomain links **TBD: cite**. DDoS attacks are an ever-present problem, as shown by recent events **TBD: cite recent events or work** and by several publications presenting new methods of mitigating DDoS attack effects **TBD: cite**. Research from Facebook, Google, and Microsoft all demonstrates considerable recent focus on providing service even under partial system failure (as partial system failure is a constant in a large networked system) [**? ? ? ?**].

Moreover, new trends in service offerings and application usage make operation under dynamic conditions both more challenging and important. Service Providers increasingly offer mission-critical services such as enterprise solutions [**? ? ? ? ?**], so ensuring performant, reliable operation is more important for these services now than ever. Recent work shows that user traffic demands are highly variable due to flash crowds, congestion, and path changes and so are much harder to plan/optimize for than inter-datacenter demands [**?**]. Operators regularly report such events on social media and blog posts **TBD: cite**. Such trends could become more salient as 5G/next-generation applications drive massive amounts of traffic from users to services [12], making user demands even more variable and more challenging to satisfy.

2.2 Approaches to Interdomain Routing

A challenge in offering low-latency, reliable services to users is routing traffic from user networks to Service Provider networks since Service Providers lack full control of which interdomain path traffic takes. BGP, the Internet's interdomain routing protocol, computes paths in a distributed fashion, giving each intermediate network a say in which paths are chosen and which are communicated to other networks. Service Providers can, however, advertise their reachability to peers/providers in different ways to increase the chance of there being good paths for users. Today, Service Providers either direct users to specific sites using `unicast` prefix advertisements [8, 14**?**], or use `anycast` prefix advertisements to provide relatively low latency and high availability at the expense of some control [1, 7**?**] **TBD: more**.

`anycast`, where Service Providers advertise a single prefix to all peers/providers at all sites, offers high availability following failures since BGP automatically ensures reachability to the deployment after tens of seconds for most networks [**?**]. Prior work shows that this availability comes with higher latency in some cases [9, 7]. `unicast` advertises a unique prefix at each site, enabling user redirection to a particular site [3]. Hence, recent work proposes hybrids of `unicast` and `anycast` which achieve a happy medium between performance and availability [1, 18**? ?**]. This latter class of solutions advertises different prefixes to subsets of all peers/providers to offer users many low-latency options across different links/sites. We refer to these solutions as `selectivecast` solutions since they are *selective* about who they advertise prefix reachability to, and have traits of both `anycast` (advertising at many sites to many peers/providers) and `unicast` (many prefixes, selectivity).

2.3 Current Approaches Do Not Consider Dynamic Conditions

In `anycast`, `unicast`, and `selectivecast` scenarios, Service Providers set capacities based on steady-state demand. For example, Service Providers might overprovision resources by a fixed percentage. Hence, none of these approaches to advertising reachability (`unicast`, `anycast`, or `selectivecast`) explicitly plan for dynamic traffic conditions. For example, `AnyOpt` and `PAINTER` (`selectivecast` solutions) find low-latency paths [18**?**], but it is unclear whether these paths can satisfy user demands under a flash crowd, and it is unclear how to scale these approaches to account for such scenarios. None of these approaches *even take user demands into account* when computing how to advertise reachability.

Instead, the current state-of-practice to handle dynamic conditions is to drain traffic from overloaded links/sites by *withdrawing* prefix announcements that are also advertised via other (healthy) links/sites [**?** 10**?**]. The traffic destined to those prefixes then arrives on other links/sites advertising those prefixes after BGP reconverges. However, this solution to handling dynamic conditions is post-hoc, unpredictable (as BGP is unpredictable), and ineffective. For example, TIPSY only achieves 70% prediction accuracy and (in a case study) required several prefix withdrawals to mitigate a single congested link [10].

Another common solution to tackling dynamic conditions is to overprovision resources to handle predicted peak loads in the future [**? ?**], but Figure 1 demonstrates that doing so can incur excessive costs. Figure 1 computes differences in peak utilization on links between different successive time periods, using longitudinal link utilization data from OVH cloud [**?**]. We split the dataset into successive, non-overlapping N-day periods and compute changes in 95^{th} percentile link utilization from one period to the next. We choose this percentile to reduce noise. Even when computing near-peak loads over successive 120-day time periods, near-peak loads can increase by 10% for 7% of links, 20% for 2% of links, and increase as high as 70%, which illustrates the amount Service Providers must overprovision to ensure they can meet resource demands. Using backup paths to handle such (rare) events can help keep costs low.

2.4 Key Problem

The key problem with current practices of routing users to Service Providers is that Service Providers have no guarantee

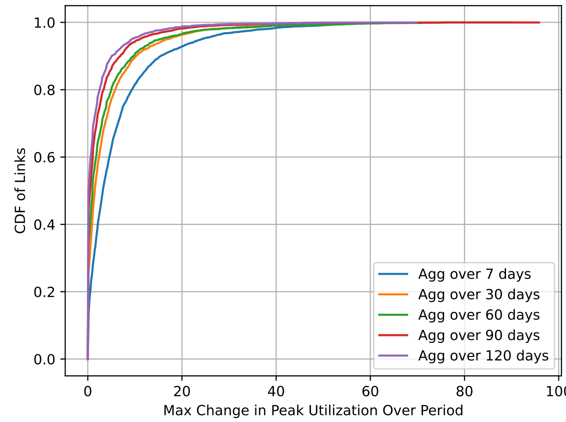


Figure 1: Planning for future peak loads requires excessive overprovisioning.

that backup paths/sites can handle new traffic loads. Prior work finds ways to prepare for intradomain failure even under dynamic traffic conditions [**? ? ? ? ?**], because in that setting the network has full *control* of how traffic flows over paths and so can ensure demand does not exceed capacity.

Compared to this intradomain setting, two key differences in the interdomain setting are that (a) a Service Provider cannot precisely control the paths traffic takes and (b) a Service Provider cannot easily determine the bottleneck capacity of paths before placing traffic on them. We explicitly address the former problem and leave the latter problem as future work. That is, we consider the problem of ensuring reliable (failure-resistant) interdomain routing when we know the available capacity of backup paths. We approximate the capacity of a path as the capacity of the corresponding peering link through which that path ingresses to the Service Providers' deployment. We consider traffic *ingressing* the deployment since, in the egress setting, the Service Provider has full control over traffic up until the egress point which is the only part of the egress path the Service Provider can control. Existing systems direct egress traffic to optimize performance and mitigate congestion [14, 17, 5].

Although we do not explicitly address problem (b) above, we implicitly address it by finding failure-resistant interdomain routing strategies (*i.e.,* having backup paths to route around congestion is a good thing), and there are natural extensions of our setting that consider such effects. For example, one could use congestion-detection mechanisms along paths to trigger traffic shifts to other, healthier (backup) paths.

Service Providers lack interdomain control for their ingress traffic, but there is hope that they can handle dynamic conditions. A Service Provider has enough global capacity to handle even large changes in traffic or sudden traffic shifts from one link to another [**?** 6]. It can use this global capacity to handle large, localized loads instead of provisioning for peak loads, but no current solution to interdomain routing makes use of this global capacity.

Figure 2 shows how a certain type of dynamic condition — failure — can lead to performance problems, even with `selectivecast` advertisements, and how Service Providers can avoid such overutilization with enough foresight. In normal operation, user traffic is split evenly across two prefixes. However when site B fails, all traffic ingresses through the same link since BGP chooses the route through Provider 1, causing overload (links have capacity 1). Advertising prefix 2 to provider 2 at site A *a priori* allows the Service Provider to split traffic between the two links during failure, avoiding overload.

Figure 2 encapsulates why current solutions to handling dynamic conditions are inefficient. `selectivecast` solutions that expose low-latency paths [**?** 18] do not plan for changing traffic conditions, only giving users low-latency *primary* paths. Withdrawing/advertising prefixes to shift load is another way to address this failure [10], but such strategies could inundate other, healthy links such as Internet routes are hard to predict. Moreover, such strategies still result in reduced performance as detecting, withdrawing, and advertising prefixes takes time. Finally, the Service Provider could overprovision to plan for failure, but doing so is a waste of resources as the deployment has enough global capacity to handle traffic congestion-free as Figure 2c demonstrates.

2.5 Key Challenges

Since Service Providers connect globally with thousands of networks, there is generally sufficient available capacity across paths/sites to satisfy user demand even under dynamic conditions. For example, an operator at a large Service Provider told us that they often have redundant connections to important customers at multiple sites. Placing traffic demand on paths to optimize performance objectives subject to capacity constraints would therefore be simple *if we could expose all the paths*.

However, exposing paths uses prefixes which are expensive [**?**]. IPv4 prefixes are monetarily expensive and pollute BGP routing tables. Prior work generally found that advertising O(50) prefixes was acceptable [18**?**], and most Service Providers advertise fewer than 50. IPv6 is not a good alternative, as such entries take 8× the amount of memory to store in a router. Since we cannot expose all the paths by advertising a unique prefix to each connected network, we must find some subset of paths to expose.

Finding that right subset of paths to expose that satisfies performance objectives, however, is hard since interdomain routing is difficult to model and since traffic conditions/failures are hard to predict. Greedy approaches to solving the problem (*e.g.,* [**?**]) do not work because every iteration of greedy allocation is tightly coupled with all other iterations due to capacity constraints on links — a decision at one iteration can negatively interfere with decisions at all other iterations. Moreover, there are many failure conditions to consider, far too many to explicitly optimize for, and under each failure condition it is unclear how users will route to the deployment due to complex interdomain routing dynamics.

3 Methodology

3.1 Problem Setup and Definitions

3.1.1 High-Level Overview

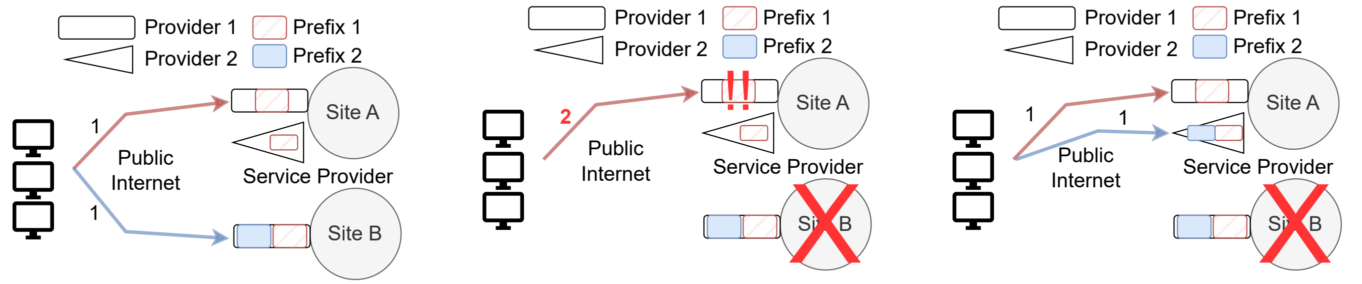
We aim to advertise relatively few prefixes to connected networks to offer low-latency paths from user networks to the Service Provider subject to capacity constraints on links. We assess resilience based on single-link and single-site failures, but our methodology extends to the multi-link/site failure case and our evaluation shows that solutions that optimize for these metrics naturally offer other benefits (§5). Adding capacity constraints on sites (*e.g.,* servers) would be a straightforward extension.

We assume the Service Provider has some technology for directing traffic towards prefixes. Examples include DNS [3, 1**?**], multipath transport [**? ?**], or control-points at/near user networks [**? ?**]. DNS offers slow redirection due to caching [**?**] but is the most readily deployable by the largest number of networks, whereas Service Provider-controlled appliances offer precise control but may not be a feasible option for some Service Providers. Service Providers with stronger incentives to provide the best service to users will invest in better options with more control, and eventually multipath transport will see wide enough deployment to be used by all Service Providers. Today, MPTCP is enabled by default in iOS [**?**] and Ubuntu 22 [**?**]. In our evaluations, we assume all users can be precisely directed to each prefix.

Service Providers connect to peers/providers at sites via physical connections we call peering links. Users route to the deployment through the public Internet to a prefix, over one of the peering links via which that prefix is advertised. The path (and therefore peering link) is chosen via BGP. We consider users at the granularity of user groups (`UG`*s*), which generally refer to user networks that route to the Service Provider similarly and experience similar latency, but could mean different things to different Service Providers (*e.g.,*/24 IPv4 prefixes, metros). `UG`*s* generate known traffic volumes, $v_{\texttt{UG}}$, and the Service Provider provisions capacity at links/sites to accommodate this load. We assume a system run by the Service Provider measures latency from `UG`*s* to prefixes advertised by the Service Provider (a reasonable assumption [2, 3, 4, 18**?**]). For a given `UG`, paths towards prefixes may be different and so may have different latencies.

3.1.2 Problem Formulation

As in prior work [18**?**], we model an advertisement configuration A as a set of $\langle$`peering, prefix`$\rangle$ pairs where



(a) Normal operation

(b) Overloaded link during failure

(c) Good operation during failure

Figure 2: In normal operation traffic is split between two sites by directing half the traffic to each prefix (2a). When site B fails, there is enough global capacity to serve all traffic (each link can handle 1 unit) but no way to split traffic across multiple providers given available paths leading to link overload (2b). A *resilient* solution is to advertise prefix 2 to an additional provider at site A, allowing traffic splitting across the two links (2c).

$\langle$`peering, prefix`$\rangle \in A$ means that we advertise that prefix via that peering. We then seek an advertisement configuration that minimizes latency (L) during normal operation (N) and single link/site failures (F_l) subject to capacity constraints, c_l:

$$\min_{A} \sum_{\mathtt{UG}} L_{\mathtt{UG}}(A, N, c_l) v_{\mathtt{UG}} + \sum_{l} \alpha_l \sum_{\mathtt{UG}} L_{\mathtt{UG}}(A, F_l, c_l) v_{\mathtt{UG}} \tag{1}$$

The weighting factors α_l control the tradeoff between a desire to minimize steady-state latency and to minimize failure-state latencies. Having different α_l for different links/sites l is not necessary, but allows operators to, for example, optimize for failures which are more likely.

Computing $L_{\mathtt{UG}}(A, N, c_l)$ requires taking into consideration not only a `UG` s' available path options and picking the lowest-latency one, but also considering other `UG` s' path options and possibly shifting `UG`*s* to higher-latency links given that many `UG`*s* share links and that those links are capacity constrained.

A given advertisement configuration exposes a set of paths to `UG`*s* through peering links (l), each of which we identify with tuples (`UG`, l). To compute the latency ($L_{\mathtt{UG}}(A, N, c_l)$) a `UG` experiences under an advertisement configuration A under deployment N or F_l (representing the deployment with a component failure l), we solve for the globally optimal allocation of `UG` traffic to paths under A. This allocation is the solution to the following linear program for $w_{UG,l}$ — the allocation of `UG` traffic to paths.

$$\begin{aligned}
\min_{w_{\mathtt{UG},l}} \quad & \sum_{\mathtt{UG},l} L_{\mathtt{UG},l}\, w_{\mathtt{UG},l} + \beta MLU \\
\text{s.t.} \quad & \sum_{l} w_{\mathtt{UG},l} = v_{\mathtt{UG}} \quad \forall \mathtt{UG} \\
& MLU \geq \frac{\sum_{\mathtt{UG}} w_{\mathtt{UG},l}}{c_l} \quad \forall l \\
& w_{\mathtt{UG},l} \geq 0
\end{aligned} \tag{2}$$

The minimization term is the sum of latency and maximum link utilization (*MLU*), weighted by a parameter β. β represents a tradeoff between using uncongested links/sites and low propagation delay and is set by the Service Provider based on their goals. We first solve *Equation* (2) with *MLU* = 1 to see if we can allocate traffic to paths with zero congestion. However such congestion-free solutions do not always exist for arbitrary advertisement strategies.

The first constraint requires that `UG` volume on a link $w_{\mathtt{UG},l}$ be non-negative and sum to the `UG`'s total volume, $v_{\mathtt{UG}}$, across all links. Constraining *MLU* to be at least as much as the utilization of each link (the second constraint) and the minimizing (a sum including) it forces *MLU* to be the maximum link utilization. Hence, optimizing Equation (2) amounts to finding low latency assignments of `UG`*s* to links that cause minimal congestion. By considering solutions to Equation (2) in both normal operations, N, and under link/site failure, F_l, we encourage low-latency, resilient advertisement strategies in Equation (1).

3.2 Approximations to Accommodate Internet Limitations

Solving Equation (1) is challenging because computing $L_{\mathtt{UG}}(A, N)$ requires advertising prefixes in the Internet, which can only be done infrequently to avoid route-flap-dampening. For example, in our optimization (detailed below) we compute $L_{\mathtt{UG}}(A, N)$ for millions of different A which could take tens to hundreds of years at a rate of advertising one strategy per hour. Hence we model, instead of measure, `UG` paths and improve this model over time through measurements.

3.2.1 Modeling `UG` Paths

Representing both Equation (1) and Equation (2) as a single mixed-integer linear program (*i.e.,* solving directly for A) would be feasible if we knew exactly how `UG`*s* routed to all possible advertisements, but routing is hard to predict [10**?**].

We instead model routing probabilistically and update our probabilistic model over time as we measure how `UG`*s* route to the deployment, adapting methods from prior work [18**?**

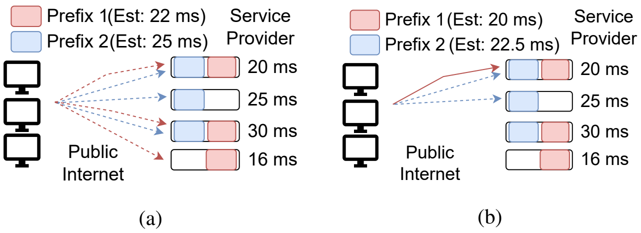


Figure 3: With no knowledge of routing preferences, we estimate latency from this UG to both the red and blue prefixes with the average over possible ingress latencies (3a). We measure towards the red prefix (*e.g.,* using `traceroute`) and learn that the first ingress has higher preference than the third and fourth ingresses. We use this information to refine our latency estimate towards the blue prefix (since the third ingress is no longer a possible option (3b)) without measuring the blue path.

?]. That is, in Equation (1) we model $L_{\mathtt{UG}}(A,N)$ probabilistically. Our probabilistic model assumes apriori, for a given UG towards a given prefix, that all ingress options for a UG are equally likely and that we know user latencies to every ingress. Upon learning that one ingress is preferred over the other, we exclude that less-preferred ingress as an option for that UG in all future calculations for all prefixes for which both ingresses are an option. As we exclude more options, latency/path distributions on unmeasured scenarios converge to the true latency/path. An example of this process is shown in Figure 3, where we refine our latency estimate towards an unmeasured prefix (blue) using measurements towards other prefixes (red).

3.2.2 Modeling Capacity-Free UG Decisions

To assign traffic to prefixes we solve Equation (2). However, for unmeasured advertisements, we only have an estimate of the distribution of possible (as opposed to actual) paths. For a given A, there are many possible paths from UGs to prefixes, so Equation (2) is too challenging to solve even probabilistically. Instead, when computing $L_{\mathtt{UG}}(A,N)$ we assume every user takes their lowest-latency path. Such decisions do not take capacity into account, which we account for later.

Since we model paths as probabilistic, user path choices and what latency UGs experience ($\sum_{\mathtt{UG}} L_{\mathtt{UG}}(A,N)v_{\mathtt{UG}}$) is also probabilistic. We model the objective function in Equation (1) as the *expected value* of the summations. That is, if the latencies for UG across different paths under advertisement A are $L_{\mathtt{UG},l}(A)$, we compute the distribution of the minimum choice, $\min_l L_{\mathtt{UG},l}(A)$, and then compute the distribution over the sum $\sum_{\mathtt{UG}} \min_l L_{\mathtt{UG},l}(A)v_{\mathtt{UG}}$. We use the minimum latency choice as a simple approximation of how traffic is directed to sites for which we can compute a probability distribution (as opposed to solving Equation (2), which we cannot do probabilistically).

3.2.3 Imposing Capacity Constraints

It could be that such minimum latency assignments lead to congested links, especially during link/site failures. We would like to penalize such advertisement scenarios, and favor those that distribute load more evenly without solving Equation (2) which would take too long. Hence, before computing the expected latency, we first compute the probability that links are congested by computing the distribution of link utilizations from the distribution of UG assignments to paths. We then inflate latency for UGs on overutilized links.

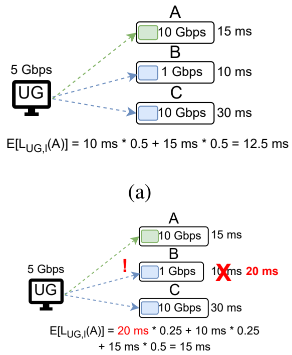
That is, for each possible outcome of UG assignments to links, we compute link utilizations and note the probability those UGs reach each link. We then accumulate the probability a link is congested as the total probability over all possible scenarios that lead to overutilization. We discourage UGs from choosing paths that are likely congested using a heuristic — we emulate the impact of congestion by inflating latency in this calculation for UGs on those paths proportional to the overutilization factor so that minimum latency path choices would push users to another path. After emulating the effect of this congestion, we recompute the distribution of average UG latency **without changing UG decisions**, and take the expected value of this random variable in Equation (1) as a proxy for the true latency. We do not change UG decisions as doing so could induce an infinite calculation loop if new decisions also lead to congestion. This heuristic penalizes advertisement strategies that would lead to many overutilized links if every user were to choose their lowest latency option. We show an example in Figure 4.

During optimization we observed that these heuristics tended to yield accurate estimates of overall benefit when averaged across the entire deployment (*i.e.,* Equation (2)), despite inaccuracies in predicting any individual UG's latency or link's utilization.

3.3 Solving with Gradient Descent

We cannot solve Equation (1) directly/exhaustively since the optimization variable is an integer matrix. Instead, we approximate the optimization variable A as a continuous variable with entries between 0 and 1 and threshold its entries at 0.5 to determine if a prefix is advertised/not advertised to a certain peer/provider. We then solve Equation (1) using gradient descent, approximating gradients between adjacent advertisements using sigmoids as in related work in the optimization literature **TBD: cite**.

Gradient descent is appropriate for two reasons. First, it is parallelizable, which is important given the high dimension of A (tens to hundreds of thousands of entries representing all ⟨`peering, prefix`⟩ pairs). Second, it simultaneously weighs several (possibly opposing) goals in Equation (1) according to their global importance in minimizing the objective. Intuitively, Equation (1) has opposing goals, since advertising



(a)

(b)

Figure 4: A UG has a path to two prefixes, green and blue. The green prefix is only advertised via ingress A and the blue prefix is advertised via two ingresses B and C, each of which are equally likely for this UG. If the UG prefers B over C, we assign the client to the blue prefix (10 ms) while if the UG prefers C over B we will assign the client to the green prefix (15 ms). Hence the initial expected latency is the average of 10 ms and 15 ms corresponding to whether this UG prefers ingress B or C (4a). However, ingress B does not have sufficient capacity to handle this UG's traffic and so is congested with 50% probability corresponding to the 50% probability that this user prefers ingress B. We artificially inflate this UG's expected latency (and all other UG's using ingress B) to reflect this possible overutilization (4b).

prefixes to expose backup paths may push some UGs off their lowest-latency route but may also encourage demand spreading during failures, which is more important than having "optimal" steady-state latency. Gradient descent takes into account both of these advantages across all ⟨peering, prefix⟩ pairs simultaneously.

We do not compute all entries of the gradient of Equation (1) since there are too many to compute. Instead, we use Monte-Carlo techniques, subsampling entries of the gradient to estimate the gradient as in prior work **TBD: cite**.

Upon thresholding entries of our optimization variable A at 0.5, we obtain an advertisement strategy to test in the real Internet. We conduct such advertisements during optimization if we have not advertised another strategy in the last hour (to avoid route flap dampening). Conducting such advertisements refines our latency estimates as they inform us of which ingresses are more preferred.

Minimization scales linearly with the number of UGs, quadratically with the number of peers/providers, and linearly with the number of prefixes. Despite this high complexity, in practice the implementation runs quickly (minutes per gradient step) relative to the rate at which we can advertise prefixes (once an hour). Although solving Equation (1) does not have convergence guarantees, we have found that it finds good solutions over a wide range of simulated topologies, and converges quickly with thousands of UGs and ⟨peering, prefix⟩ pairs (§5).

3.4 Exploration to Improve Our Model

As, apriori, we do not know UG preferences, we may be uncertain about the benefit of other advertisement strategies which could make solving our model using gradient descent noisy/unstable. For example, an adjacent strategy could have better *expected* latency than the current one but have much worse *actual* latency. Gradient descent would then push us towards a worse strategy.

To improve convergence, we periodically compute benefit distributions on adjacent advertisement schemes and look for strategies for which we are very uncertain about whether it will be better or worse. One could use different measures of uncertainty — we choose entropy. Testing advertisement strategies in the real Internet takes time (as measuring all paths takes time) but we observed that it improves convergence.

4 Implementation

We prototype SCULPTOR on the PEERING testbed [?], which is now available at 32 Vultr cloud sites. We describe how we built SCULPTOR on the real Internet and how we *emulate* a Service Provider including their clients, traffic volumes, and resource capacities. (We are not a Service Provider and so could not obtain actual volumes/capacities, but our extensive evaluations (§5) demonstrate SCULPTOR's potential in an actual Service Provider and our open/reproducible methodology provides value to the community.)

4.1 Simulating Clients and Traffic Volumes

To simulate client performances, we measured actual latency from IP addresses to our PEERING prototype as in prior work [?] and selected targets according to assumptions about Vultr cloud's client base.

We first tabulate a list of 5M IPv4 targets that respond to ping via exhaustively probing each /24. Vultr informs cloud customers of which prefixes are reachable via which peers, and we use this information to tabulate a list of peers and clients reachable through those peers.

After tabulating peers, we then measure latency from all clients to each peer individually by advertising a prefix solely to that peer using Vultr's BGP action communities and pinging clients from Vultr. We also measure performance from all clients to all providers individually, as providers provide global reachability.

In our evaluations, we limit our focus to clients who had a route through at least one of Vultr's direct peers (we exclude route server peers [?]). Vultr likely peers with networks with which it exchanges a significant amount of traffic [14], so clients with routes through those peers are more likely to be "important" to Vultr. We found 700k /24s with routes through 1086 of Vultr's direct peers. In an effort to focus on interesting optimization cases, we removed clients whose lowest latency to Vultr was 1 ms or less, as these were assumed to be addresses related to infrastructure, leaving us with measurements from 666k /24s to 825 Vultr ingresses.

As we do not have client traffic volume data, we simulate traffic volumes in an attempt to both balance load across the deployment but also encourage some diversity in which clients have the most traffic. To simulate client traffic volumes, we first randomly choose the total traffic volume of a site as a number between 1 and 10 and then divide that volume up randomly among clients that anycast routes to that site. Client volumes in a site are chosen to be within one order of magnitude of each other.

Although these traffic volumes are possibly not realistic, we believe that by demonstrating the efficacy of `SCULPTOR` over a wide range of subsets of sites and simulated client traffic volumes, we demonstrate that `SCULPTOR`'s benefits are not tied to any specific choice of sites or traffic pattern within those sites.

4.2 Deployments

We use a combination of real experiments and simulations to evaluate `SCULPTOR`. Both cases use simulated client traffic volumes, but our real experiments measure real paths using RIPE Atlas probes, while our simulations use simulated paths.

4.2.1 Experiments in the Internet

We assess how `SCULPTOR` performs in the real Internet using RIPE Atlas probes [15], which represent a subset of all clients. RIPE Atlas allows us to measure paths (and thus ingress links) to prefixes we announce from PEERING, which `SCULPTOR` needs to refine its model (**??**). However, RIPE Atlas does not have large coverage, as probes are only in 3,000 networks, and we are limited by RIPE Atlas probing rate constraints. Choosing RIPE Atlas probes to maximize coverage of networks and countries, we select probes from **TBD: X** networks which have paths to **TBD: Y** ingresses.

4.2.2 Simulations

We also evaluate `SCULPTOR` by simulating user paths. Simulating user paths allows us to conduct more extensive evaluations as experiments take less time and use clients in more networks. To limit computational complexity, we select 100 clients with a route through each ingress. Service Providers could similarly limit computational complexity by optimizing over, for example, a certain percent of the traffic which is a common practice in the networking optimization literature [1**? ? ?**]. Over all studied deployments, we consider paths from clients in 42k /24s to 868 peers/providers.

4.3 Setting Resource Capacities

We assume that resource capacities are overprovisioned proportional to their usual load. However, we do not know the usual load of links and cannot even determine which peering link traffic to one of our prefixes arrives on, as Vultr does not give us this information. (This limitation only exists since we are not a Service Provider, as a Service Provider could measure this using IPFIX, for example to measure steady-state link loads.) We overcome this limitation using two methods (corresponding to our two deployments — Section 4.2), each with their pros and cons.

For our first method of inferring client ingress links, we advertise prefixes into the Internet using the PEERING testbed [**?**], and measure actual ingress links to those prefixes using traceroutes from RIPE Atlas probes [15]. Specifically, we perform IP to AS mappings and identify the previous AS in the path to Vultr. This approach has limited evaluation coverage, as RIPE Atlas probes are only in a few thousand networks. In cases where we cannot infer the ingress link even from a traceroute, we use the closest-matching latency from the traceroute to the clients' (known) possible ingresses. For example, if the traceroute's latency was 40 ms to Vultr's Atlanta site and a client was known to have a 40 ms path through `AS1299` at that site, we would say the ingress link was `AS1299` at Atlanta.

The second method we use to determine ingress links is simulating user paths by assuming we know all users' *preference models* (§3.2). We use a preference model where clients prefer peers over providers, and clients have a preferred provider. When choosing among multiple ingresses for the same peer/provider, clients prefer the lowest-latency option. We also add in random violations of the model. This second approach allows us to evaluate our model on all client networks but may not represent actual routing conditions. However, we found that our key evaluation results (§5) hold regardless of how we simulated routing conditions (we also tried completely random routing), suggesting that our methodology is robust to such assumptions. Prior work also found the preference model to be valid in 90% of cases they studied [18].

Given either method of inferring client ingress links we then measure paths to an anycast prefix and assign resource capacities as some overprovisioned percentage of this catchment. (Discussions with operators from CDNs suggested that they overprovision using this principle.) We choose an overprovisioning rate of 30%.

5 Evaluation

`SCULPTOR` compares favorably to all other solutions on all studied dimensions. In particular, it yields an advertisement strategy that gives clients lower latency paths and fewer congested links during both steady-state and failure. Hence, SCULPTOR uses the same routing resources (prefixes, sites, links) more efficiently than other solutions, improving performance and reducing cost.

Note that in the following evaluations we do not know the optimal solution for a given budget of prefixes and instead use One-per-Peering as a proxy for an oracle comparison (which has a large, unreasonable prefix budget). Hence, differences between each advertisement strategy and One-per-Peering represent an upper bound of each solutions' suboptimalities, as we do not know if advertisement strategies that achieve One-per-Peerings' properties with fewer prefixes exist.

5.1 General Evaluation Setting

We compare SCULPTOR's ability to give users low-latency paths to the Service Provider, subject to capacity constraints both during steady-state and failure conditions, to that of other advertisements strategies. The strategies include

anycast: A single prefix announcement to all peers/providers at all sites. Pure anycast is a very common strategy used in today's deployments.

unicast: A single prefix announcement to all peers/providers at each site. Another common strategy used in today's deployments [8, 3].

AnyOpt/ PAINTER: Two strategies proposed in the literature for reducing steady-state latency compared to anycast.

One-per-Peering: Advertise a unique prefix to each peer/provider, so all possible backup paths are always available to users. This solution provides a performance upper-bound, even though it is prohibitively expensive. (We obviously do not know the actual optimal solution with fewer prefixes.)

We conduct evaluations both on the public Internet (*i.e.,* with real routing dynamics, Section 5.2) and in simulation (with simulated user path preferences, Section 5.3). Evaluating SCULPTOR in the public Internet demonstrates that SCULPTOR provides real, tangible benefits, but these evaluations take longer than simulated ones as advertising prefixes is slow. Evaluating SCULPTOR using simulated routing dynamics allows us to conduct micro-evaluations in a variety of hypothetical scenarios.

For our evaluation in the public Internet, we **TBD: fill in details**.

For our simulated evaluations, we compute solutions over many random routing preferences, demands, and subsets of sites to demonstrate that SCULPTOR's benefits are not limited to a specific deployment property. We evaluate SCULPTOR over deployments of size 3, 5, 10, 15, 20, 25, and 32 sites. Each size deployment was run at least **TBD: 10** times with random user demands and routing preferences. We use twice the base-2 logarithm of the number of peers/providers as the number of prefixes for all solutions (except **One-per-Peering**). For deployments with more sites we use approximately 70 prefixes, and for smaller ones we use between 10 and 30. Prior work found that using this many prefixes to improve performance was a reasonable cost [18**?**].

In solving Equation (2), the lowest link utilization solution may have overloaded resources in failure scenarios. We say all traffic arriving on a congested link is congested, and do not include this traffic in quoted latency comparisons (actual latency would be a complicated function of congestion control protocols and queueing behavior). We comment separately on how much traffic is congested.

5.2 Internet Deployment Results

Keepin it real

5.3 Simulation Results

We find SCULPTOR to provide improvements over prior work in all studied metrics, but that these relative improvements often decrease as the number of sites increases. We compute both average overall latency and the fraction of traffic within 10 ms, 50 ms, and 100 ms of the One-per-Peering solution for each advertisement strategy. We investigate these latency thresholds in particular since they correspond to the recommended network performance of highly immersive (*e.g.,* VR/AR), immersive (*e.g.,* gaming), and lower-immersive (*e.g.,* web browsing) applications [11], respectively. Hence, quantifying the fraction of traffic within these latencies of the One-per-Peering solution isolates how much the lack of control over interdomain routing in particular affects the Service Provider's ability to satisfy application requirements. (As opposed to say, for example, how the placement of sites affects the Service Provider's ability to satisfy application requirements.) For all advertisement strategies, we compute traffic to path allocations (thus latencies) using Equation (2).

5.3.1 SCULPTOR Finds Good Steady-State Paths

We first assess SCULPTOR's ability to find low-latency paths for users in scenarios without failure. Figure 5 demonstrates that SCULPTOR finds lower latency paths for more traffic than any other solution. The average latency for SCULPTOR compared to One-per-Peering ranges from 0 to 3 ms while the next-best solution (PAINTER) achieves between 3 and 5 ms compared to One-per-Peering.

To generate Figure 5 we compare user latencies for each advertisement strategy to user latencies with the One-per-Peering advertisement. Anycast comparatively performs the worst with users being on average **TBD: 25.3 ms** worse than One-per-Peering whereas SCULPTOR performs the best with users being **TBD: 6.4 ms** worse than One-per-Peering. Interestingly, unicast (**TBD: 14.9ms** worse than One-per-Peering) actually performs better than AnyOpt (**TBD: 17.4ms** worse than expensive) which could be due to the different settings (AnyOpt was designed to optimize over a small number of provider connections).

These average latency benefits translate into differences in the fraction of traffic that can use different types of applications. SCULPTOR has between **TBD: nums** of traffic within 10 ms of the One-per-Peering solution, **TBD: nums** within 50 ms, and **TBD: nums** within 100 ms. These percentages compare to the next-best solution, PAINTER, which has **TBD: nums**. Even differences of a few percent are sigiificant, as they could represent large numbers of users in a global-scale deployment.

Providing lower steady-state latency than all other solutions was not an explicitly stated goal of SCULPTOR, but happens anyway, demonstrating the power of SCULPTOR's efficient path modeling and optimization methodology.

5.3.2 SCULPTOR Improves Resilience to Ingress/Site Failure

We next assess SCULPTOR's ability to provide resilience to ingress failures and site failures which models any case where a link/site is no longer available. Examples of such failures include excessive DDoS traffic on the link/site (thus using the link/site as a sink for the bogus traffic), physical failure, and/or planned maintenance. Figure 6 demonstrates that SCULPTOR effectively shifts traffic without overloading alternate links/sites and without excessively penalizing performance, improving the fraction of traffic within 10 ms of One-per-Peering, compared to the next-best solution, by **TBD: nums** pct.

To generate Figure 6 we fail each ingress/site once and compute UG to link allocations using Equation (2). For each failed component and advertisement strategy, we compute the difference between achieved and One-per-Peering latency for users who use that component in their steady-state solution. **TBD: congestion fig** shows the fraction of traffic that lands on congested links for each solution. We do not include congested traffic in average latency computations (Fig. 6a, Fig. 6e) but say such traffic does *not* satisfy 10 ms, 50 ms, or 100 ms objectives.

TBD: Anycast comparatively performs the worst with users being on average TBD: 25.3 ms worse than One-per-Peering whereas SCULPTOR performs the best with users being TBD: 6.4 ms worse than One-per-Peering. Interestingly, unicast (TBD: 14.9ms worse than One-per-Peering) actually performs better than AnyOpt (TBD: 17.4ms worse than expensive) which could be due to the different settings (AnyOpt was designed to optimize over a small number of provider connections).

These average latency benefits translate into differences in the fraction of traffic that can use different types of applications. SCULPTOR has between TBD: nums of traffic within 10 ms of the One-per-Peering solution, TBD: nums within 50 ms, and TBD: nums within 100 ms. These percentages compare to the next-best solution, PAINTER, which has TBD: nums. Even differences of a few percent are sigiificant, as they could represent large numbers of users in a global-scale deployment.

It is interesting that both PAINTER and unicast provide good resilience relative to SCULPTOR, or at least better than we expected, especially as the number of sites increases. Speculating on why these solutions provide decent resilience, we note that unicast exposes one path to all sites for all users, and so we expect that, on average, most users will have at least one decent backup path, even if that backup path is not the best one. Similarly, PAINTER tries to expose the best path for every user. We hypothesize that this path exposure randomly exposes good backup paths for most users, even though PAINTER did not consider that benefit in its allocation process.

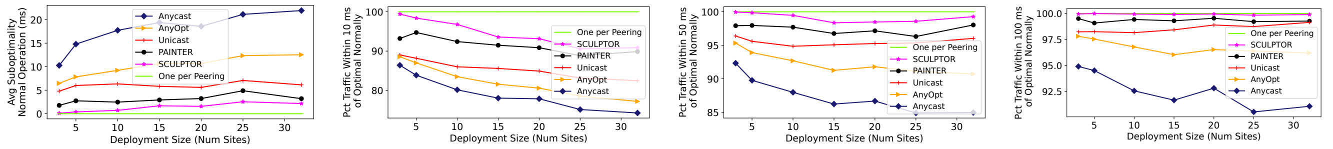
5.3.3 SCULPTOR Improves Infrastructure Utilization

One response to increased link/site utilization is to install more capacity so that there is sufficient headroom to satisfy peak demand. Figure 7 shows that this response is not always necessary with better routing — SCULPTOR finds ways of distributing load over existing infrastructure to accommodate increased demand without adversely affecting latency. We quantify this improved infrastructure utilization by showing how much extra peak load each routing strategy can help satisfy without congestion, where peak load is distributed according to two realistic traffic patterns: flash crowds and diurnal effects.

We define a flash crowd as a momentary traffic increase at a single site. Examples of flash crowds might include content releases (games, software updates) that spur downloads in a particular region or popular streamed events like the super bowl/world cup. Since increased demand is highly localized, we can spread excess demand to other sites if paths to those sites exist. Figure 7a shows that SCULPTOR absorbs up to a **TBD: 175**% increase in localized traffic before any link experiences congestion.

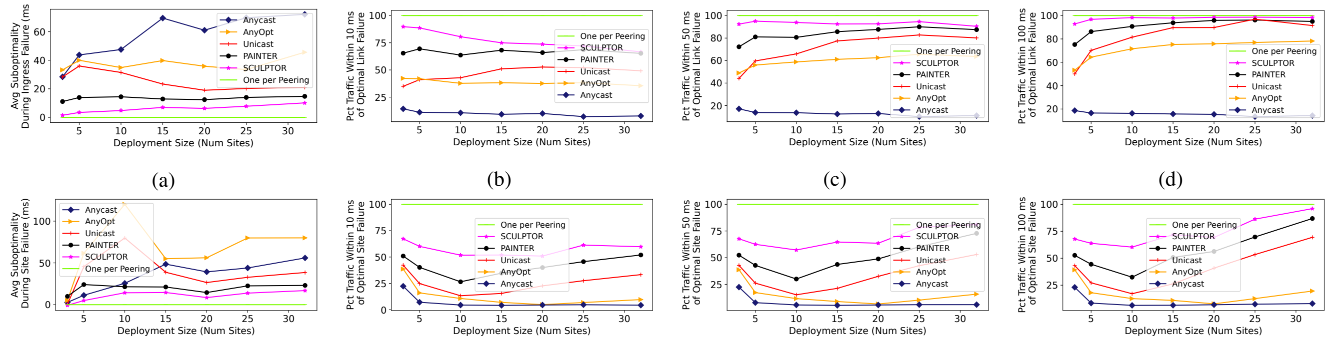
To generate Figure 7a, we identify each client with a single "region" (*i.e.,* site) corresponding to the site at which they have the lowest possible latency ingress link. For each region individually, we then scale each client's traffic in that region by M% and compute traffic to path allocations using Equation (2). We increase M until any link experiences congestion when we scale volume in any region. For example, if Atlanta experiences congestion with a 60% increase in traffic for Atlanta clients, but no other region does, we stop at that critical value of $M = 60\%$. Figure 7a plots the average of such critical M values over simulations for each deployment size.

Our diurnal analysis in Figure 7b uses a similar methodology. We define a diurnal effect as a traffic pattern that changes volume according to the time of day. Diurnal effects might be different for different Service Providers, but a prior study



(a) (b) (c) (d)

Figure 5: `SCULPTOR` lowers average latency and increases the fraction of traffic within critical thresholds of their best-case latency during normal operation.



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 6: `SCULPTOR` lowers average latency and increases the fraction of traffic within critical thresholds of their best-case latency during both link and site failures.

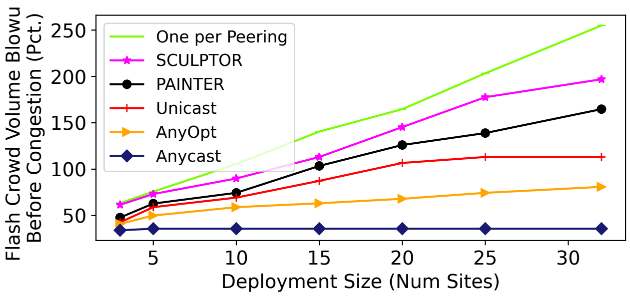
from a large Service Provider demonstrates that these effects can cause large differences between peak and mean site volume [**?**]. We sample diurnal patterns from that publication and apply them to our own traffic. We group sites in the same time zone, and assign traffic in different time zones different multipliers — in "peak" time zones we assign a multiplier of M and in "trough" time zones a multiplier of 0.1 M (the full diurnal curve we use is in **TBD: appendix!!**). As in our flash crowd analysis, we increase M until at least one link experiences congestion at at least one hour of the day and plot the average such critical value of M over simulations for each deployment size.

Both Figure 7a and Figure 7b that `SCULPTOR` finds ways to route more traffic without congestion than other solutions. `SCULPTOR` can handle single-site flash crowds at **TBD: 175**% more than the expected volume for deployments with 32 sites, creating an almost 2× savings in link costs assuming Service Providers plan for peak loads. `SCULPTOR` also handles more intense diurnal traffic swings, routing traffic with **TBD: 20**% more intense swings than both `PAINTER` and `unicast`. Hence, instead of scaling deployment capacity to accommodate peak time-of-day traffic, Service Providers can re-distribute traffic to sites in off-peak time zones.

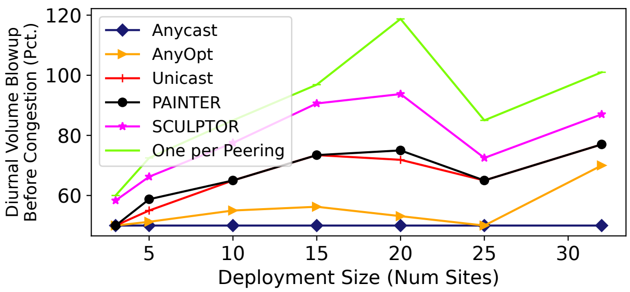
Prior work noted similar benefits by satisfying user requests in distant sites [**?**], but moved traffic on their own backbone to realize these gains. `SCULPTOR`'s benefits are orthogonal to this prior work, and useful for Service Providers without private backbones, as they use the public Internet to realize the same result. It is interesting that `PAINTER` provides similar benefits to `unicast`— this result is likely since `PAINTER` only aims to reveal low-latency primary paths for users, whereas `unicast` guarantees reachability to all sites guaranteeing users a minimum number of paths (one of which hopefully has capacity).

6 Related Work

- Other/related solutions to the problem
  - Painter, tipsy, anyopt, fastroute, footprint, jiangchen, analyzing the performance of an anycast cdn
- Egress
  - Espresso, swan, edge-fabric
- Resilience
  - Hose-based cross-layer backbone network design with Benders decomposition
    - ∗ Capacity-efficient and uncertainty-resilient backbone network planning with hose (obsolete tho)
  - EBB: Reliable and Evolvable Express Backbone Network in Meta
  - Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure
  - Decentralized cloud wide-area network traffic engineering with {BLASTSHIELD}
- Congestion / problems on paths
  - Zooming in on Wide-area Latencies to a Global Cloud Provider



(a)



(b)

Figure 7: SCULPTOR improves infrastructure utilization so that Service Providers can underprovision compared to peak loads.

  - ∗ Significant fractions of performance problems are in the "middle" of the path, and thus potentially fixable by switching to a different path
- – Anycast Agility: Network Playbooks to Fight DDoS
  - ∗ Manually craft set of BGP actions to place certain volumes of traffic on certain sites

- 5g/multipath systems
  - – Evolving Mobile Cloud Gaming with 5G Standalone Network Telemetry
  - – Xlink

7 Conclusions

In conclusion

References

[1] Matt Calder, Ashley Flavel, Ethan Katz-Bassett, Ratul Mahajan, and Jitendra Padhye. 2015. Analyzing the Performance of an Anycast CDN. In *IMC*.

[2] Matt Calder, Ryan Gao, Manuel Schröder, Ryan Stewart, Jitendra Padhye, Ratul Mahajan, Ganesh Ananthanarayanan, and Ethan Katz-Bassett. 2018. Odin: Microsoft's Scalable Fault-Tolerant CDN Measurement System. In *NSDI*. 501–517.

[3] Fangfei Chen, Ramesh K Sitaraman, and Marcelo Torres. 2015. End-user mapping: Next generation request routing for content delivery. In *SIGCOMM*.

[4] Wouter B De Vries, Ricardo de O. Schmidt, Wes Hardaker, John Heidemann, Pieter-Tjerk de Boer, and Aiko Pras. 2017. Broad and load-aware anycast mapping with verfploeter. In *IMC*.

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

[6] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. 2013. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*.

[7] Thomas Koch, Ethan Katz-Bassett, John Heidemann, Matt Calder, Calvin Ardi, and Ke Li. 2021. Anycast in context: A tale of two systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*.

[8] Rupa Krishnan, Harsha V Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. 2009. Moving Beyond End-to-End Path Information to Optimize CDN Performance. In *SIGCOMM*.

[9] Zhihao Li, Dave Levin, Neil Spring, and Bobby Bhattacharjee. 2018. Internet Anycast: Performance, Problems, & Potential. In *SIGCOMM*.

[10] Michael Markovitch, Sharad Agarwal, Rodrigo Fonseca, Ryan Beckett, Chuanji Zhang, Irena Atov, and Somesh Chaturmohta. 2022. TIPSY: predicting where traffic will ingress a WAN. In *SIGCOMM*.

[11] Nitinder Mohan, Lorenzo Corneo, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, and Jussi Kangasharju. 2020. Pruning edge research with latency shears. In *HOTNETs*.

[12] Larry Peterson and Oğuz Sunay. 2020. 5G Mobile Networks: A Systems Approach. *Synthesis Lectures on Network Systems* (2020).

[13] Brandon Schlinker, Todd Arnold, Italo Cunha, and Ethan Katz-Bassett. 2019. PEERING: Virtualizing BGP at the Edge for Research. In *CoNEXT*.

[14] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. 2017. Engineering egress with edge fabric: Steering oceans of content to the world. In *SIGCOMM*.

[15] RIPE NCC Staff. 2015. RIPE Atlas: A Global Internet Measurement Network. *Internet Protocol Journal* (2015).

[16] VULTR. 2023. VULTR Cloud. vultr.com/

[17] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. 2017. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *SIGCOMM*.

[18] Xiao Zhang, Tanmoy Sen, Zheyuan Zhang, Tim April, Balakrishnan Chandrasekaran, David Choffnes, Bruce M Maggs, Haiying Shen, Ramesh K Sitaraman, and Xiaowei Yang. 2021. AnyOpt: predicting and optimizing IP Anycast performance. In *SIGCOMM*.