Contra: A Programmable System for Performance-aware Routing

Kuo-Feng Hsu
Rice University

Ryan Beckett
Microsoft Research

Ang Chen
Rice University

Jennifer Rexford
Princeton University

Praveen Tammana
Princeton University

David Walker
Princeton University

Abstract

We present Contra, a system for performance-aware routing that can adapt to traffic changes at hardware speeds. While point solutions exist for a fixed topology (e.g., a Fattree) with a fixed routing policy (e.g., use least utilized paths), Contra can operate seamlessly over any network topology and a wide variety of sophisticated routing policies. Users of Contra write network-wide policies that rank network paths given their current performance. A compiler then analyzes such policies in conjunction with the network topology and decomposes them into switch-local P4 programs, which collectively implement a new, specialized distance-vector protocol. This protocol generates compact probes that traverse the network, gathering path metrics to optimize for the user policy dynamically. Switches respond to changing network conditions by routing flowlets along the best policy-compliant paths. Our experiments show that Contra scales to large networks, and that in terms of flow completion times, it is competitive with hand-crafted systems that have been customized for specific topologies and policies.

1 Introduction

Configuring a network to achieve a diverse range of objectives, such as routing constraints (*e.g.*, traffic should go through a series of middleboxes), and traffic engineering (*e.g.*, minimize latency and maximize throughput), is a challenging task. To handle this complexity, one approach has been to use SDN solutions, which have a centralized point for management [25, 26]. However, centralized controllers are inherently too slow to respond to fine-grained traffic changes, such as short traffic bursts. In fact, even the software control planes locally on the switches are often limited in their ability to select new routes fast enough.

Recent work has developed load-balancing mechanisms that operate entirely in the data plane to enable real-time adaptation [11, 30]. By making use of fine-grained performance information on hardware timescales, these systems can deliver considerable performance benefits over static load-balancing mechanisms like ECMP. Unfortunately, existing systems, such as Conga [11] and Hula [30], are point solutions that only work under specific assumptions about the network topology, routing constraints, and performance objectives—they only support a "least utilized shortest path" policy on a data center topology. It is not obvious how to adapt them for other kinds of topologies or policies.

In this paper, we describe Contra, a general and programmable system for performance-aware routing. Network operators configure Contra by describing the network topology as well as a high-level policy that defines routing constraints and performance objectives. Contra then generates P4 programs for switches in the network, which execute in a fully distributed fashion. Collectively, they implement a specialized version of a distance-vector protocol that forwards traffic based on routing constraints and optimizes for the user-defined performance objectives. This protocol operates by generating periodic probes that traverse policy-compliant paths and collect user-defined performance metrics. Switches analyze the incoming probes and rank paths in real time, storing the current best next hop to reach any given destination. Since the programs run in the data plane, switches can react to performance changes quickly. Overall, Contra is designed to achieve the following objectives:

- General – operates over a wide range of policies
- Reusable – works correctly for any topology
- Distributed – does not require central coordination
- Responsive – adapts to changing metrics quickly
- Implementable – on today's programmable data planes
- Policy-compliant – packets only use allowed paths
- Loop-free – mitigates persistent/transient loops
- Optimal – converges to best paths under stable metrics
- Stable – mitigates oscillation under changing metrics
- Efficient – avoids undue traffic and switch overhead
- Ordered – limits out-of-order packet delivery

To achieve these objectives, we need to address several challenges. First, to operate over arbitrary topologies, Contra requires new techniques to search the set of possible paths for optimal routes. State-of-the-art solutions, such as Conga [11] and Hula [30], assume a tree-based data center topology, which makes exploring possible paths, avoiding forwarding loops, and finding optimal routes straightforward. Second, link and path metrics can change constantly, which may

Objective	Key idea(s)	Section(s)
General	Language for performance-aware routing Policies as path-ranking functions	2
Reusable	Policy analyzed jointly with topology	4.1
Distributed Responsive & Implementable	Synthesis of data-plane routing protocol Periodic probes to collect path metrics Implemented in P4	4.1–4.3
Policy-compliant	Probes and packets carry policy states Switches keep track of state transitions	4.1–4.3
Loop-free	Monotonicity analysis Probes carry version numbers Early loop breaking for flowlets	2, 5.1, 5.5
Optimal Stable & Efficient	Isotonicity analysis Limit the frequency of probes Failure detection and metric expiration	2, 5.2, 5.4
Ordered	Policy-aware flowlet switching	5.3

Figure 1: Key ideas in Contra.

cause unsynchronized views at different switches. Making forwarding decisions based on inconsistent views may lead to forwarding loops or paths that violate the routing policy. Third, a naïve solution that constantly changes routes can cause transient or even persistent chaos. We draw inspirations from wireless network routing [16, 38, 39], and design mechanisms that leverage programmable data planes to address this. Finally, we develop policy-aware flowlet switching, which routes flowlets to mitigate out-of-order packet delivery while ensuring policy compliance.

Summary. We make several contributions in the design of Contra, and Figure 1 summarizes the key ideas.

- We define a new programming abstraction that views policies as path-ranking functions, and generalizes existing languages by allowing operators to specify path constraints and dynamic metrics simultaneously.
- We design a new configurable, performance-aware, distance-vector routing protocol.
- We develop compilation algorithms that generate switch-local P4 programs that implement a particular configuration of the protocol based on user policy.
- We have built a system prototype, and conducted thorough experiments to demonstrate that Contra is competitive with state-of-the-art systems that are customized for a specific topology and routing policy.

Non-goals. There has been abundant recent research on efficient load-balancing strategies, especially in data centers. The goal of this work is not to outperform such strategies in the contexts for which they have been manually optimized. Rather, our goal is to facilitate the deployment of such techniques on a much broader set of networks and with a broader collection of optimization criteria, and to do so without asking network operators to take the time, or acquire the expertise necessary, to write "assembly-level" P4 programs.

2 Policy language

Contra includes a high-level language that can express a wide range of user policies, which are functions that rank network paths. Our compiler then ensures that switches always use the best policy-compliant paths. Users can combine regular expressions, which express hard constraints on the allowed paths, with performance metrics to express dynamic preferences. As a concrete example, consider the following policy:

```
minimize( if A .* then path.util else path.lat )
```

It first classifies paths using a regular expression (`A .*`), and then based on the classification, it defines the rank to be either path utilization or latency. Each node will separately choose its best paths according to this function. So node A will always choose the least utilized path, while all other nodes will select the path with the lowest latency.

The Contra language can also capture static policies in existing systems that are not related to performance. For instance, FatTire [40] uses regular expressions to classify legal and illegal paths (though it says nothing about the *performance* of such paths). To route packets through a waypoint `W`, a FatTire policy would be `(.* W .*)`, which allows any path through `W` but no other paths. Contra can represent this by mapping all legal paths to 0 and illegal paths to ∞:

```
minimize( if .* W .* then 0 else ∞ )
```

This policy will ensure that every node always selects a path through `W` if one exists in the network, and drops traffic otherwise; no path is preferred to a path with rank ∞.

As another example, Propane [14] allows users to write policies about failover preferences. A Propane policy `(A B D) >> (A C D)` indicates a preference for sending traffic through path `A B D` and only using `A C D` if the first path is not available (*e.g.*, a link has failed). In Contra, we can achieve the same effect by ranking paths statically as below.

```
minimize( if A B D then 0 else if A C D then 1 else ∞ )
```

In Contra, it is also possible to rank paths based on multiple metrics. For example, suppose we prefer that A reaches D via B instead of via C, and we also prefer shorter, less utilized paths. This can be achieved by lexicographically ranking paths, *e.g.*, prefer paths through B first, then shortest paths, and finally, least utilized paths.

```
minimize( if A .* B .* D then (0, path.len, path.util)
     else if A .* C .* D then (1, path.len, path.util)
     else ∞ )
```

Ranking paths using regular expressions defines strict, inviolate preferences; however, operators may have softer constraints based on path performance: *e.g.*, one path may be preferred up to a point, but if the utilization is too high then some traffic should be shunted along another path instead. For example, to prefer least-utilized paths when the network load is light (utilization of the path is less than 80%), even if those paths are long, but to prefer shortest paths when network load is heavy (and hence to save bandwidth globally), one might use the following policy.

```
minimize( if path.util < .8
          then (1, 0, path.util)
          else (2, path.len, path.util) )
```

Policy

pol	::=	minimize(e)	*optimization*

Expressions

e	::=	n	*constant numeric rank*
	|	∞	*infinite rank*
	|	path.attr	*path attribute*
	|	$e_1 \circ e_2$	*binary operation*
	|	if b then e_1 else e_2	*if statement*
	|	$(e_1, \ldots, e_n)$	*tuple*

Boolean Tests

b	::=	$r \mid e_1 \leq e_2 \mid$ not $b \mid b_1$ or $b_2 \mid b_1$ and b_2

Regular Paths

r	::=	node_id $\mid$. $\mid r_1 + r_2 \mid r_1\, r_2 \mid r^*$

Figure 2: Syntax for Contra policies.

Finally, to steer traffic towards or away from particular links, one may add or subtract weights. For instance, the following policy demonstrates how to add weight to costly links AB and CD while otherwise using simple shortest paths.

```
minimize( (if .* AB .* then 10 else 0) +
          (if .* CD .* then 20 else 0) + path.len )
```

Figure 2 presents the full language syntax, and Table 1 presents selected policy examples taken from the literature. The key novelty of the language is that it can capture many of the *static* conditions expressed by earlier work such as FatTire [40] or NetKAT [13] as well as the *relative* preferences of Propane [14], and yet it also augment such policies with *dynamic* preferences based on current network conditions.

Policy	Implementation
P1. Shortest path routing [24]	path.len
P2. Minimum utilization [30]	path.util
P3. Widest shortest paths [32]	(path.len, path.util)
P4. Waypointing [13]	if .*(F_1+F_2).* then path.util else ∞
P5. Link preference [14]	if .*XY.* then path.util else ∞
P6. Weighted link [19]	(if .*XY.* then 10 else 0) + path.len
P7. Source-local preference [12]	if X.* then path.util else path.lat
P8. Congestion-aware routing [27]	if path.util $<$ .8 then (1, 0, path.util) else (2, path.len, path.util)

Table 1: Selected Contra policies.

Policy analysis and guarantees. Contra requires user policies to be *monotonic* (metrics do not improve for longer paths) and *isotonic* (switches have consistent preferences). If a policy is non-isotonic (*e.g.*, P8), Contra will attempt to decompose it into multiple isotonic subpolicies that can be processed separately. Contra can do this for many conditional policies (*e.g.*, P8), but it will not always succeed, e.g., for "shortest widest paths"; see Appendix A for more discussion. These algebraic constraints guarantee that when metrics are stable, new flows will be sent along globally optimal paths [22]. Our system also guarantees that hard constraints expressed by regular expressions are never violated. Under changing metrics, when switches make distributed decisions based on their local views, routes may be suboptimal [11].

3 Selected Challenges

Contra addresses three key challenges. To illustrate these challenges, we first describe a simple strawman solution designed for a specific topology (data center networks) and specific policy (use least utilized paths). Consider the simple leaf-spine topology in Figure 3(a), where switch S wants to send traffic to switch D over the least-utilized path:

```
minimize( if S.*D then path.util else ∞)
```

One strawman solution is to use a distance-vector protocol, where each switch propagates link metrics (*i.e.*, utilization) to its neighbors via periodic probes, and builds up a local forwarding table of "best next hops" to reach other switches.

Concretely, at time 1, D sends two probes to A and B carrying utilizations u(A-D)=0.1 and u(B-D)=0.2, respectively. Upon receiving a probe, a spine switch updates its metric, and then disseminates the probe to its downstream neighbors. The updated probe metric is the maximum of a) the original probe metric, and b) the utilization of the inbound link from the switch's neighbor, so the probe always carries the utilization of the bottleneck link on its traversed path. For instance, when B receives the probe from D, it updates the utilization to 0.3, which is the maximum of a) the original probe metric, u(B-D)=0.2, and b) the utilization u(S-B)=0.3; when A receives the probe from D, it updates the utilization in the probe to be 0.4, which is the maximum of u(A-D)=0.1 and u(S-A)=0.4. At time 2, both A and B disseminate the updated probes to S. Now, S has received probes on both paths S-A-D (u=0.4) and S-B-D (u=0.3), and it chooses B as the best next hop to reach D due to its lower utilization. Changes in link metrics are then propagated by the next round of probes. In fact, this describes Hula [30], a state-of-the-art solution for utilization-aware routing in data centers.

Challenge #1: Arbitrary topologies. On a tree topology, simple mechanisms (*e.g.*, defining a set of "downstream" and "upstream" neighbors for each switch) suffice to explore paths and prevent forwarding loops [30], but on a non-hierarchical topology, it is insufficient.

Consider the sequence of events in Figures 3(b)-(e), where S prefers the least-utilized path to D. Suppose that at time 1, D sends out probes to A and S, and A propagates D's probe to B and S, with the utilizations shown in Figure 3(b); now, both B and S prefer to reach D via A. At time 2, S propagates A's probe to B about S-A-D (u=0.1), so B changes its preference to go through S; B then propagates S's probe to A (u=0.2), but it gets delivered only at time 4. At time 3, u(A-D) increases to 0.5, which is discovered by a new periodic probe from D to A and S. From A's perspective, the best path to reach D is still A-D, except that now the utilization is 0.5 instead. At time 4, when B's (old) probe to A arrives with u=0.2, A mistakenly thinks that it should instead reach D via B, not knowing that A is itself on B's best path to reach D. As a result, a forwarding loop S-A-B-S would form, and it will *persist* as long as the link utilizations remain stable.

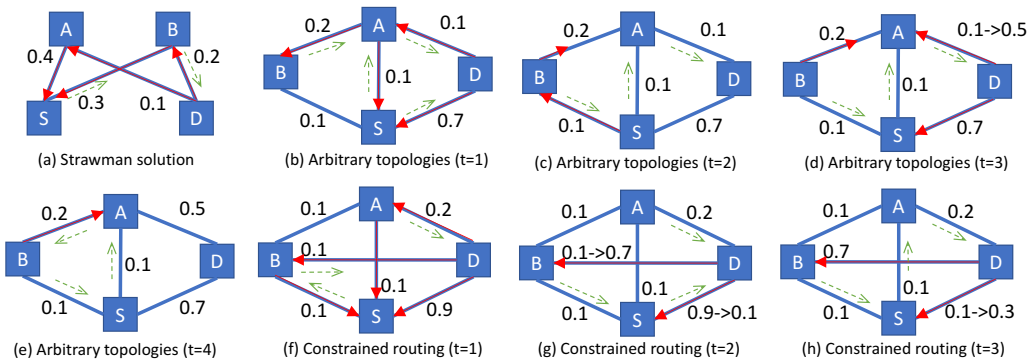


Figure 3: Supporting sophisticated policies over arbitrary topologies is challenging. (Solid, red arrows represent probes, and dotted, green arrows represent packet forwarding. Links are labeled with performance metrics.)

It might seem that path-vector protocols would address this problem, where probes record their traversed paths, and switches avoid picking paths that involve themselves. However, the root cause of transient loops is the inconsistent views during network convergence; so transient loops can still form even with path-vector protocols [37]. Carrying the path traversed by the probe would also increase traffic overheads and the complexity for processing probes.

Solution. Our solution is inspired by DSDV [39] and a more recent proposal Babel [16], which were originally developed for wireless mesh networks. At a high level, switches assign version numbers to probes, so that they can identify and avoid using outdated probes. In addition, Contra uses flowlet switching [44] to pin traffic to particular paths and avoid out-of-order packet delivery. Still, because flowlet entries expire at different times, it is possible for transient loops to form on rare occasions. Contra quickly detects and breaks such loops by monitoring hop counts.

Challenge #2: Constrained routing. Supporting routing policies with path constraints leads to additional challenges. Consider the scenario in Figure 3(f), where the policy is not only to prefer least-utilized paths, but also that traffic should never first go through B and then A due to security concerns:

```
minimize(if .*B.*A.* then ∞ else path.util)
```

Under this policy, S can only send traffic to D via a) S-D, b) S-A-D, c) S-B-D, or d) S-A-B-D; initially, S prefers c) (u=0.1). Now consider the sequence of events shown in Figures 3(f)-(h). Suppose that at time 1, the traffic from S arrives at B. At time 2, the u(B-D) increases to 0.7, and u(S-D) decreases to 0.1, so B updates its best next hop (to reach D) to be S, preferring the path B-S-D. At time 3, B sends the traffic back to S, which already forms a loop. But things can get even worse: at time 3, u(S-D) increases to 0.3, so S changes its preference to be S-A-D (u=0.2). So the traffic has been forwarded along a path S-B-S-A-D, which not only contains a loop but also violates the intended policy.

Solution. Contra compiles the regular expression constraints in the user policy into automata, and intersects these automata with the network topology to obtain a *product graph* [45, 14], which specifies a probe and packet tagging scheme for each switch. Intuitively, tags represent states of the user-defined automata; by checking that probes and packets carry the right state when arriving at a switch, it is possible to enforce the global user policy in a distributed fashion. This tagging scheme guarantees that no packet ever deviates from a user's regular expression constraints in the policy.

Challenge #3: Custom performance metrics. Supporting custom performance metrics also introduces new challenges. As discussed earlier, a switch only propagates the probe with the *best* metric to its neighbors. However, such local decisions do not always give rise to globally optimal results, unless the policy is *isotonic* [22] (*i.e.*, roughly speaking, downstream nodes respect the preferences of upstream nodes). Unfortunately, some useful policies, such as some congestion-aware routing schemes, are not isotonic [27].

Solution. Contra analyzes the user policy to determine if it is isotonic. If not, Contra decomposes the non-isotonic policy into multiple isotonic subpolicies. Information about each subpolicy is propagated separately in different classes of probe and the best probe from each class is chosen locally. The classes are recombined and a route corresponding to the best current path is chosen only at a traffic source. Hence, if metrics are stable, then new flows will be sent along globally optimal paths. To avoid packet reordering due to unstable metrics, we follow Conga and Hula's strategy and use flowlet switching, which trades the fact that packets in pinned flowlets may follow suboptimal paths for stability.

4 Compilation: Stable metrics

The goal of the compiler is to generate a particular configuration of the Contra protocol that efficiently implements the desired policy in the data plane. We describe compilation in two phases. First, in this section, we describe an algorithm that operates *as if link metrics do not change*, so probes only need to be propagated once. The next section explains how this algorithm is extended to handle changing metrics.

Challenge. One key challenge during compilation involves

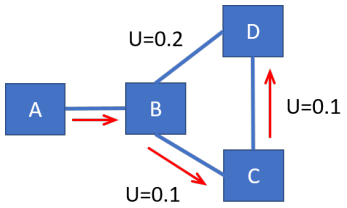


Figure 4: Naïve solutions may lead to suboptimal paths. Node A uses ABCD even though a better path ABD exists.

policies with conditional regular expression matches, such as `(if r then m1 else m2)`, because nodes may rank paths differently based on the branch of the conditional they use. In fact, conditional regular expression matches are one source of non-isotonicity: if every node selects the best next hop according to its own preferences alone, other nodes might wind up with suboptimal routes. For example, consider the following policy when applied to the topology in Figure 4:

```
minimize( if (A B D) then 0 else path.util)
```

In this example, A prefers path ABD, but B prefers the least utilized path BCD. The correct behavior in this scenario would be for B to carry A's traffic along path ABD while simultaneously sending its own traffic along path BCD.

However, a naïve (and erroneous) implementation may disseminate probes along the paths DB and DCB[1] and ask B to decide which path is best. In this case, B would use the probe from DCB and discard the one from DB. However, if the latter probe is discarded, A will not receive information about its preferred route! To avoid this, another naïve solution would be to propagate probes along all possible paths in the network to avoid missing good paths. For instance, B might send every probe it receives to A. However, this would lead to far too many probes, as the number of paths in a graph may be exponential in the number of nodes.

Solution. Instead, for a conditional `(if r then m1 else m2)`, if one could determine the path with minimal metric m1 that matches r using one probe, and separately determine the path with minimal metric m2 that does not match r using another probe, then nodes could delay choosing their best path until both probes have been received and only then combine the information to make a decision. This is one concrete instance where Contra needs to decompose the non-isotonic policy (due to regular expressions) into multiple isotonic subpolicies. Contra achieves this by creating an efficient data structure that combines all regular expressions appearing in a policy with the network topology, and by sending separate probes for different regular expression matches.

4.1 Finding policy-compliant paths

Inspired by Merlin [45] and Propane [14], Contra constructs a data structure called a *product graph* (PG), which compactly represents all paths allowed by the policy.

[1] Recall that probes travel in the opposite direction to actual traffic.

Policy automata. A policy's regular expressions define the different ways the shape of a path can affect its ranking. To process a policy, we first convert all such regular expressions into finite automata. Because probes disseminate information starting from the destination, but policies describe the direction of traffic that flows in the opposite direction, we actually construct an automaton for the reverse of each regular expression. Each automaton is a tuple $(\Sigma, Q_i, F_i, q_{0_i}, \sigma_i)$. Σ is the alphabet, where each character represents a switch ID in the network. Q_i is the set of states in automaton i. The initial state is q_{0_i}. F_i is the set of accepting / final states. $\sigma_i \colon Q_i \times \Sigma \to Q_i$ is the transition function. Consider the example policy in Figure 5(b), which a) allows A to reach D via the path A-B-D, b) allows B to reach D via any path with the least utilization, and c) disallows all other paths. The Contra compiler would generate the automata in Figure 5(c).

Network topology. The construction of the automata has not considered the actual network topology, so not all automaton transitions are legitimate. For instance, although the automaton for `D.*B` could in principle accept a sequence of transitions `D-A-B`, this sequence would never happen on the network shown in Figure 5(a), simply because D is not directly connected to A. Therefore, our compiler merges the topology with the automata and prunes invalid transitions.

Product graph (PG). If there are k automata (one for each regular expression used in the policy), then each state in the PG would have $k+1$ fields, $(X, s_1, \cdots, s_k)$, where the first field X is a topology location, and s_i is a state in the i-th automaton; there is a directed edge from $(X, s_1, \cdots, s_k)$ to $(X', s'_1, \cdots, s'_k)$, if a) $X - X'$ is a valid link on the topology, and b) for each automaton i, we have $\sigma_i(s_i, X') = s'_i$.

Concretely, in the PG in Figure 5(d), every edge represents both valid transitions on the two policy automata and a valid forwarding action on the topology. As examples, no edges exist from any `(D,*,*)` state to `(A,*,*)` state, because they have been eliminated due to topology constraints; also, there is a transition between node D0 and B0 because a) the topology connects D and B, and b) applying B to each automaton from state 1 leads to state 2. We use the symbol "–" to denote the special "garbage" state—the state from which there is no valid transition in an automaton.

Virtual nodes. We distinguish PG nodes ("virtual nodes") from topology locations ("physical nodes"). A physical node X may have multiple virtual nodes, because probes could arrive via different paths, and reach different automaton states as a result. For instance, the physical node B has two virtual nodes `(B0,-,2)` and `(B1,2,2)`; we have labeled their location fields as B0, B1 to capture this, and we call them *tags*. If multiple virtual nodes exist, then probes must be duplicated to traverse paths that satisfy different constraints. For instance, B will receive a probe for B0 representing a path on the way to matching regex ABD, and a second probe for B1 representing a path on the way to matching regex `B.*D`.

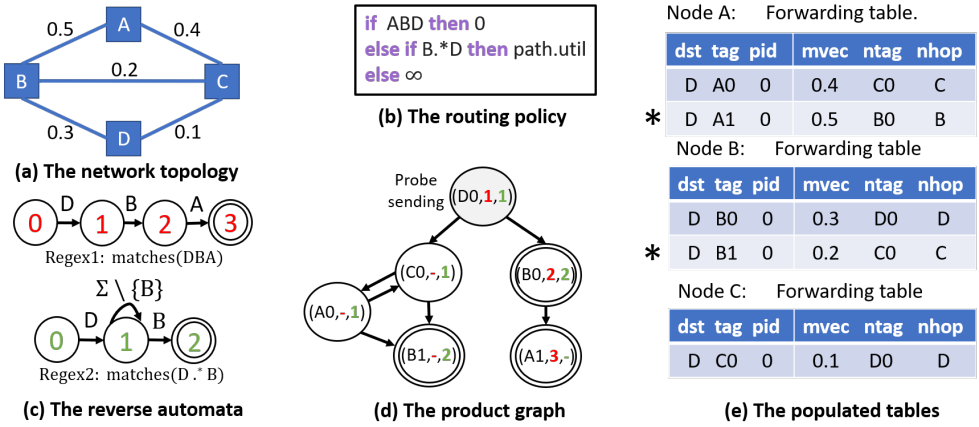


Figure 5: A running example of the compilation algorithm.

Probe sending states. If a physical node X is a valid destination allowed by the policy, then exactly one of its virtual nodes is a *probe sending* state. This state has the form $(X0, \sigma_0(q_{0_0}, X), \cdots, \sigma_k(q_{0_k}, X))$; all probes that originate from X initially carry this state. This is because, when probes start at the originating node, they have only traversed the first hop "X" from the initial automata states q_{0_i}.

Policy compliance. Any path in the PG from an accepting state to a probe sending state is a policy-compliant path. All policy-compliant physical paths also exist in the PG.

4.2 Packet forwarding

Before describing the protocol itself, we first describe the structure of the forwarding (`FwdT`) tables on each switch. The compiler only generates the table layout, and then actual entries are populated at runtime based on link metrics, which we describe in the next subsection.

An entry in the forwarding table has the form `[dst*,tag*,pid*,mv,ntag,nhop]`, where the star fields are table lookup keys. Each row of the table indicates where the given switch will send packets destined for `dst` when those packets carry a PG node `tag` and probe number id `pid`. The sender of packets will set the initial tag and the probe number based on its best path. At each intermediate hop, when a packet with a given `dst`, `tag`, and `pid` matches an entry in `FwdT`, the switch looks up the next tag (`ntag`) and replace the packet's tag with it; it also forwards the packet to the next hop (`nhop`). The metrics vector (`mv`) is not used for packet forwarding, but for populating the entries. A property of `FwdT` is that any `tag`-`ntag` pair in this table corresponds to a PG edge, and when a `ntag` is written into a packet it is then forwarded out the `nhop` port that leads to a topology node corresponding to that `ntag`. This process implies that forwarding will always follow edges in the PG.

As an example, consider the `FwdT` table for switch `B`: the policy allows `B` to reach `D` either through a) `B-D`, satisfying (part of) the regular expression `ABD`, or through b) the best of `B-D`, `B-C-D`, and `B-A-C-D`, satisfying the regular expression `B.*D`. The former corresponds to the virtual node `B0` in the PG, and the latter is implemented by a combination of both `B0` and `B1`. Hence, the reader may observe that it is possible for nodes of the product graph to contribute to the implementation of more than one regular expression in the policy—this sharing improves algorithm performance as a single probe can contribute to uncovering information useful in more than one place in the policy.

Ignoring for now how the forwarding entries were populated, consider the first entry in `B`'s table in Figure 5(e). The entry is generated from the virtual node `B0`: if a packet is at `B` with `tag=B0` and a destination `D`, then either that packet was sent from `A`, and traveled to `B` or it was sent directly from `B`. In either case, the current best path is through the next hop `nhop=D` with a metric `mv=0.3`. Moreover, before `B` sends the packet to `D`, it should update the tag to the new virtual node's tag `D0`. The second entry in `B`'s table is generated from `B1`. When packets are tagged with `B1`, there are two paths they could take to `D`: `B-C-D` and `B-A-C-D`. Currently, the least utilized path is `B-C-D`, so `nhop=C` and `mv=0.2`. The updated tag will then be `C0`. For this policy, only one probe is needed (carrying utilization), so there is only a single probe id (`pid`) of 0. The asterisk next to the `B1` entry indicates that `B` prefers `B-C-D` over `B-D`, which is determined after evaluating the user policy on both paths. Hence, traffic sourced from `B` will choose `B-C-D`. Note that each source can determine its own preference: although `B` prefers `C`, `A` can still use `A-B-D` since `A`'s traffic will be forwarded using the `B0` entry.

Function SwiForwardPkt in Figure 6 summarizes the packet forwarding logic. When a packet first arrives at the switch from a host, it is treated differently. In this case, this first switch must determine the preferred path for the packet (with each path having a representative destination, PG start node and probe id), which is stored in the `BestT` table.

4.3 Sending probes

While the forwarding tables compactly encode how devices should forward traffic in a policy-compliant way, we have yet to describe how these tables are populated. To this end,

```
function INITPROBE(PGNode n, ProbeId pid)
    if n.isPrbSendingState then
        p.origin ← TOTOPONODE(n)
        p.pid ← pid
        p.tag ← n.tag
        p.mv ← INITMVEC
        MULTICASTPROBE(n, p)

function MULTICASTPROBE(PGNode n, Probe p)
    pg_neighbors ← GETPGOUTNEIGHBORS(n)
    topo_neighbors← TOTOPONODES(pg_neighbors)
    MULTICAST(p→topo_neighbors)

function PROCESSPROBE(Switch S, Probe p)
    n ← NEXTPGNODE(S, p.tag)
    p.mv ← UPDATEMVEC(p.inport)
    key ← (p.origin, n.tag, p.pid)
    (mv, ntag, nhop) ← FwdT[key]
    if f(p.pid, p.mv) < f(p.pid, mv) then
        FwdT[key] ← (p.mv, p.tag, p.inport)
        oldKey ← BestT[p.origin]
        if s(key) < s(oldKey) then
            BestT[p.origin] ← key
        p.tag ← n.tag
        MULTICASTPROBE(n, p)

function SWIFORWARDPKT(Packet p, Switch S)
    key ← (p.dst, p.tag, p.pid)
    if fromHost(p.inport) then
        key ← BestT[S]
        p.pid ← key.pid
    (mv, ntag, nhop) ← FwdT[key]
    p.tag ← ntag
    SENDPKT(p, nhop)
```

Figure 6: Pseudocode for the synthesized switch-local programs. Underlined variables are PG states.

the Contra compiler generates protocol logic for propagating probes from probe sending states in order to populate the tables with the best paths to each destination.

At a high level, each node in the PG propagates probes to its neighbors. For instance, a probe starts at `D0` (`D` with tag 0) and is sent to `B0` and `C0`. `C0` updates the utilization to be 0.1 and adds this entry to its forwarding table before sending a new probe to `A0` and `B1`. Similarly, `B0` adds an entry for the probe it received from `D0` with utilization now 0.3 before sending a new probe to `A1`. `A1` receives a probe from `B0` and adds an entry with utilization 0.5, etc. `A0` receives a probe from `C0` with metric now 0.4 and adds this entry to its table before sending the probe to `C0` and `B1`. Probes will continue to propagate through the PG so long as they decrease the best available metric for that probe type and PG node. Since a static analysis ensures that policy metrics are monotonically increasing, probes will not be propagated endlessly in loops.

To determine which entry to use for forwarding local traffic, switches compute the best path by keeping a pointer to their overall best entry (the asterisks in Figure 5(e)). For example, consider the node `A`. Evaluating the policy on `A0` results in ∞ because `A0` is not an accepting state for regex `ABD` or `B.*D`. On the other hand, evaluating the policy in `A1` results in 0 (the best rank) because `A1` is an accepting state for regex `ABD`. Hence, the asterisk appears by `A1`.

Probe generation. Probes are generated from initial PG states (*e.g.*, `(D0,1,1)` in our example). These sending states use the procedure in INITPROBE to initiate probes, and use MULTICASTPROBE to multicast the probes along the outgoing PG edges to all downstream neighbors. Each probe carries four fields: (1) *origin* denotes the topology location of the sending switch (*i.e.*, `D` for the state `(D0,1,1)`); (2) `pid` is the probe id, as obtained from the policy decomposition; (3) `mv` denotes the metrics vector used in the policy (*i.e.*, utilization in the example, which is initialized to a default value 0); and (4) *tag* denotes the id of the PG node the probe is at.

Probe dissemination. The PROCESSPROBE algorithm describes how a switch processes a probe from its neighbor. It first obtains the PG node for the neighbor (n). Next, it updates the metrics in the probe based on the port at which the probe arrived, *e.g.*, the maximum of the probe's carried utilization and the local port's utilization. If this probe (with id i and tag t) contains a better metric according to f than what is currently associated with i and t, then it updates its `FwdT` table with the new `nhop`, `ntag`, and `mv` based on this probe. The switch also checks if an update affects its overall best choice (*i.e.*, where the asterisk points to), as recorded in the `BestT`. The switch looks up the existing value and compare it to the current probe using the function s that checks the overall value of the probe (not just per tag / probe id). Finally, the probe tag is updated to the correct value for n, and the probe is multicast to all PG neighbors.

5 Compilation: Unstable metrics

Consider using the same solution as described in Section 4, but instead of sending just one probe, sending many probes periodically, one per time interval. This introduces new complications due to the lack of synchronization; certain parts of the network may be working with outdated information. In fact, the example sequence from Section 3, Figure 3(b)-(f) demonstrates exactly how a problem can arise—the example culminates with the forwarding loop S-A-B-S. Notice also that this loop is technically policy-compliant because any path from S to D is allowed, so the packet tagging mechanism would not prohibit it.

The key issue is that when switches use old probes to make decisions, loops can form. In Figure 3(b), the probe p from B to A took a long time to propagate; by the time p arrived at A, the metrics had already changed again. Concretely, p was computed using an old metric u(A-D)=0.1, which had since changed to 0.5; but A still used this outdated probe and thought D was a better next hop.

5.1 Preventing persistent loops

To prevent loops, we draw on ideas from Babel [16], which distinguishes outdated probes from new ones using a version number, and discards outdated probes. In our scenario, this suggests A should discard p because it has an older version number, and should continue to use D as the next hop, thereby avoiding the loop. When a round of probes is still in propagation, switches may have temporarily inconsistent views, so a packet may experience a transient (yet policy-compliant) loop. However, versioned probes would guarantee that persistent loops would not form [16].

We note that there is a long body of work on loop prevention in routing protocols with tradeoffs being made in terms of space overhead and convergence time. Contra's compilation algorithm can potentially be integrated with different loop prevention techniques. For example one could prevent loops by adding a bit vector to each probe to record visited nodes (*i.e.*, a path-vector protocol) at the cost of greatly increased probe overhead (one bit for every router). We opt for our approach to limit the space overhead of probes.

Refinement (Versioned probes). *As before, except that a) switches attach version numbers to the probes, which increase for each round; b) the* `FwdT` *table records the version number of the probe that was used to compute each entry; and c) before a switch updates an entry with version v with a probe of version v', it needs to check that $v' \geq v$.*

5.2 Probe frequency

Versioning the probes, however, leads to an additional complexity: a node may not always be able to pick the best path. Consider a case where D sends probes to S every 0.2 ms along two available paths: a) p_1 with utilization of 0.4 and a latency of 0.1 ms, and b) p_2 with utilization of 0.1 but a latency of 0.2 ms. Due to the higher latency of p_2, whenever S receives a probe from this path, it would find the probe to be outdated, since newer probes had arrived from p_1. As a result, S ends up always using p_1 which has a higher utilization, even if the policy prefers the least-utilized path p_2.

We observe that this problem can be addressed by ensuring (with high probability) that old probes are fully propagated throughout the network before new probes are sent out. In the above scenario, if we set the probe period to be 0.2 ms or larger, then S would instead pick p_2 to be the better path after both probes have been received.

Refinement (Limited probe frequency). *As before, except that the probe period needs to be larger than or equal to $0.5 \times RTT$, where RTT is the highest round-trip time between any pair of switches in the network.*

5.3 Policy-aware flowlet switching

Since Contra can spread traffic in the same flow across multiple paths, it is important to mitigate the potential out-of-order packet delivery. One classic approach is *flowlet switching* [44], where packets in the same flow are grouped in bursts/flowlets and the same forwarding decision is applied to the entire flowlet. By doing so, the first packet in the flowlet is always forwarded to the best path, and subsequent packets in the same flowlet would inherit this (slightly outdated) forwarding decision. This also increases network stability: although each switch's best path is constantly fluctuating, at any given point, much of the current network traffic is pinned to a particular path. Only new flowlets will make use of the current path information.

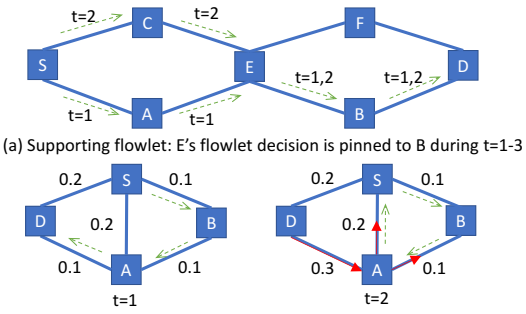


Figure 7: Challenges due to flowlet switching.

A first attempt to implement flowlet switching in Contra would be to have each switch maintains a table of the form `[fid*,nhop,t]`, where `fid` is the flowlet ID (from hashing a packet's five tuple), `nhop` is the temporarily "pinned" next hop, and `t` is the timestamp of the last packet in `fid`. When the next packet in `fid` arrives, the switch computes the gap between its timestamp and `t`: if the gap is small, this packet will use the current `nhop`; otherwise, the switch expires this entry and starts a new flowlet. Perhaps surprisingly, deploying such a flowlet switching mechanism with Contra may result in policy violations. Consider the example in Figure 7(a), where the policy prefers the least utilized of the upper or lower paths, but avoids the "zigzag" path.

```
if SCEFD + SAEBD then path.util else ∞
```

Suppose that at t=1, S sends traffic to D via the lower path due to its lower utilization; using flowlet switching, all switches temporarily pin this flowlet to their respective next hops along the path when they receive the first packet in the flowlet (*e.g.*, A pins to E at t=1.1, which expires at t=2.1; E pins to B at t=1.2, which expires at t=2.2; and so forth). At t=2, S discovers that the utilization of the upper path has improved, and changes its preference to D instead. However, if the packets from S arrive at E before t=2.2, which is its flowlet switching expiration time, E will continue to forward these packets to the lower path, causing a policy violation.

The fundamental reason for this is that flowlet switching is oblivious to routing constraints. Our solution makes it *policy-aware* by adding PG tags to flowlet entries. Concretely, policy-aware flowlet switching extends the table format to be `[tag*,pid*,fid*,nhop,t]`, where `tag` and `pid` are obtained from the probe that created the forwarding entry, and `tag`, `pid`, and `fid` are match keys. This enables flowlet switching *within each policy constraint and probe type*. Now, when E processes the packet at t=2.2, it would see that the packet was constrained to traverse the upper path and use the flowlet entry for that path.

Refinement (Policy-aware flowlet switching). *As before, except that switches perform policy-aware flowlet switching by maintaining multiple entries for the same flowlet, each for a different path constraint/tag and probe type.*

5.4 Handling failures

Switches also need to discover new best paths when links or switches fail. Suppose that the best path for S to reach D is S-A-D, but the link A-D goes down at some point. We need to ensure that S will learn about the failure and change to another available path if one exists. Our solution is to first detect failed links, and then to expire flowlet entries when their next hop is along a link that is believed to be failed.

Refinement (Expiration). *As before, except that a flowlet entry is expired when a packet arrives at a switch and is going to be forwarded by the flowlet entry, and the next hop is along a failed link.*

Handling failures, of course, requires the existence of a failure detection mechanism. The specific link failure detection methods are beyond the scope of Contra; the above approach merely ensures that switch routes around detected failures for future flowlets. In our implementation of Contra, a switch marks a link as failed when there have been no probes along the link for k probe periods, where k is a parameter that determines how fast failures should be discovered.

5.5 Breaking transient loops

As we discussed, transient loops may still occur when probes are in propagation. Figure 7(b) is a concrete example. At t=1, the best path for S to reach D is S-B-A-D. Then, at t=2, A receives a probe from D carrying a worse metric, so it propagates the probe to S and B. Before this probe arrives at S and B, A learns of the better path through S, and traffic that is already in flight will be forwarded along a transient loop S-B-A-S; this loop will be broken once S and B receive the new probe because it has a higher version number.

Interestingly, flowlet switching may lengthen the duration of transient loops because flowlet switching decisions may expire at different times across hops. Suppose that A's timer expires at t=3, and it starts using the new best next hop S to reach D; however, the timers at S and B do not expire until t=4. Then the traffic would continue to be forwarded in the loop S-B-A-S regardless of the newer probe, until S and B have updated their flowlet switching decisions.

We address this by detecting loops lazily and flushing the offending flowlet switching entries upon detection. Concretely, each switch maintains a *loop detection* table $\{$`flow_hash`*`,maxttl,minttl`$\}$, which maps a flow's CRC hash to the maximum and minimum TTL values seen at this switch. δ=`maxttl-minttl` should be stable in the absence of loops: it is the difference between the longest and the shortest paths packets could have traversed to reach the current switch. However, when there is a loop, δ would continue to grow. Therefore, switch detects a potential loop (with false positives) when its δ exceeds a threshold. When this happens, the switch expires its flowlet switching decision, and starts a new flowlet using the latest metric in the `FwdT` table. Hence, we arrive at our final solution below.

Final solution. *As before, except that switches use loop detection tables to detect and break loops by refreshing their flowlet switching decisions using the latest metrics.*

6 Evaluation

We aim to answer three main questions in our evaluation: a) How well does Contra scale to large networks? b) How competitive is Contra compared to hand-crafted systems? and c) How well does Contra work on general topologies? Due to space constraints, some results appear in the Appendix.

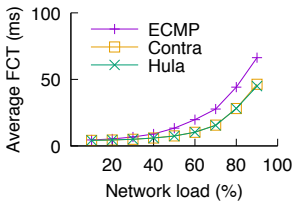
6.1 Prototype implementation and setup

Our prototype [3] consists of 7485 lines of code in F# [6], which processes policy and topology descriptions, and generates switch-local P4 programs. The compilation also minimizes the number of tags and forwarding table sizes.

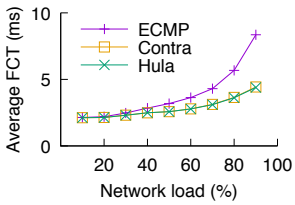
Experimental setup. We have used three types of topologies: a) data center topologies, b) random graphs, and c) real-world topologies (e.g., Abilene [1] and those from Topology Zoo [7]). Our baseline systems for data center networks are ECMP and Hula [30], load-balancing schemes for a Fattree topology. ECMP balances traffic randomly without considering network load, and Hula is load-aware and always chooses the least-utilized path among all shortest paths. Our baseline system for arbitrary graphs is SPAIN [35], which statically (*i.e.*, independently of network load) selects multiple paths to route traffic. We used two workloads obtained from production networks for our evaluation: a web search workload [12], and a cache workload [43].

Simulation vs. Emulation. For simulation, we have used a customized version of ns-3 [4] that implements P4 switches using the `bmv2` model, and it runs on a Dell server with six Intel i7-8700 CPU cores and 16 GB of RAM. For emulation, we have used the `bmv2` switches in a Mininet [34] cluster on 15 CloudLab [17] servers, each with eight cores and 128 GB of RAM. The simulation allows us to conduct experiments at high link speeds (up to 40 Gbps), as traffic forwarding is simulated in an idealized environment. The emulation, on the other hand, generates and forwards real network traffic across machines. This allows us to evaluate the systems in a high-fidelity environment [34], albeit with lower link speeds (50 Mbps) to avoid causing bottlenecks in the software switches.

We have replicated the emulation setup (45 switches in a 6-ary Fattree) in our simulator, and measured for eight setups (workloads+policies) the Pearson correlation coefficient (PCC) [42] between the emulated and simulated results. Seven of the setups produced PCC values of 0.99+, and the other produced 0.98 (1.0 means perfect correlation). This "meta" experiment confirms that the simulation and emulation results closely mirror each other, and that our observations are consistent across setups. Below, after presenting compiler results (§6.2), we first describe our simulation results (§6.3-§6.5), and then the high-fidelity emulation (§6.6).

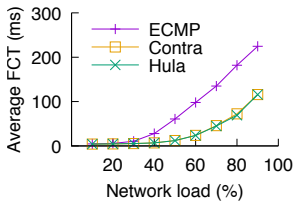


(a) The web search workload

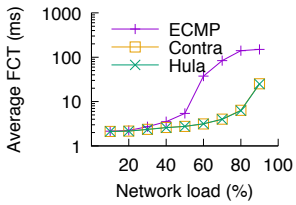


(b) The cache workload

Figure 8: Contra achieves a similar FCT as Hula, outperforming ECMP considerably.



(a) The web search workload



(b) The cache workload

Figure 9: Contra achieves a significantly shorter FCT than ECMP on an asymmetric topology with a failed link.

6.2 Compiler scalability

To test the scalability of our compiler, we used topologies of varying sizes from 20 to 500 nodes. For each topology, we evaluated three different policies: a) minimum utilization (MU: no regular expressions, single performance metric), b) waypointing (WP: three regular expressions, single performance metric), and c) congestion-aware routing (CA: no regular expression, non-isotonic policy with two performance metrics). The concrete numbers are included in Figure 14 in Appendix C, and we summarize the key takeaways here: The compiler scales roughly linearly with topology size, and completes in seconds on topologies with hundreds of nodes. Use of regular expressions increases product graph size and hence compilation time. Non-isotonic policies add some overhead due to the additional policy analysis.

We have also measured the switch state used by the generated P4 programs (Figure 15 in Appendix C). At a high level, WP and CA require more state than MU: WP's regular expressions require tracking automaton states. and CA's non-isotonic policy requires a separate table for each metric in the decomposed policy (*i.e.*, separate entries for different `pid` values). However, no more than 70 kB of switch state was necessary in any experiment—a tiny fraction of the available memory on modern switches (tens of megabytes) [2].

6.3 Performance: Data center topology

We compare Contra with ECMP and Hula in terms of their flow completion time (FCT) in simulation. In our topology (Figure 13a), we used 32 hosts with 10 Gbps links, with 10 Gbps links between switches, and an oversubscription ratio of 4:1. Half of these hosts were configured as senders, and the other half receivers. We set the probe period to 256μs and flowlet timeout to 200μs for both Contra and Hula. All links have a queue buffer size of 1000 MSS by default. Moreover, we tuned the desired network load from 10% to 90% by adjusting the flow arrival times, and obtained the FCT for each setting. The policy used in Contra is widest shortest paths (WSP; policy P3 in Table 1), which picks the least-utilized shortest paths and is equivalent to Hula; we found that the performance of Contra with the MU policy (P2 in Table 1) is similar to WSP in this setup.

Symmetric Fattrees. Figure 8 shows that both Contra and Hula outperform ECMP considerably because they balance traffic based on network load. At 90% load, they reduce the average FCT by 30% for the web search dataset and by 47% for the cache dataset. Hula outperforms Contra slightly, by 0.33% on average across different datasets and network loads. This is because Hula knows statically what paths are shortest paths (and hence what ports to send probes from), whereas Contra has to discover this information dynamically (*i.e.*, by carrying the path length as well as the utilization, and also by sending probes both "up" and "down" at each level in the datacenter)—hence Contra sends more probes than Hula in order to achieve generality over different topologies and policies. Further compiler optimizations could likely reduce this gap further (*e.g.*, by identifying shortest paths statically). We also refer interested readers to Appendix I for detailed breakdown of probe and tag overheads.

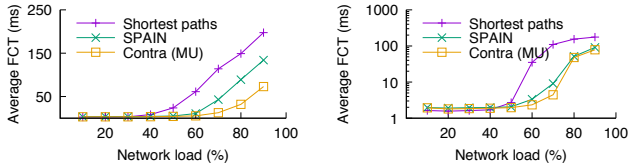
Asymmetric Fattrees. Next, we ran the same experiment after injecting a failure on a link between an aggregation switch and a core switch, so that the topology became asymmetric. Figure 9 shows the FCT for this setting. In this case, we found that ECMP incurred heavy traffic loss beyond 50% network load, even though 75% of all capacity remains after the link failure. The average FCT increased by 3.18× for the web search dataset and 8.72× for the cache dataset. In contrast, Contra and Hula only had an increase of 1.80× for the web search dataset and 1.67× for the cache dataset, relative to the FCTs on the symmetric topology.

We further measured the queue growths under ECMP and Contra with 60% workload on the web search dataset without bounding the maximum queue sizes (see Figure 16 in Appendix D). We found that Contra's queue lengths never exceeded 1000 MSS, whereas ECMP saw queue lengths larger than 1000 MSS more than 97% of the time, which can cause heavy traffic loss when the queues are full.

In order to measure Contra's response time to link failures, we sent UDP workloads at 4.25 Gbps rate, and brought down an aggregate-core link. Contra successfully detected the link failure after 800μs, which is close to the failure detection threshold (3×probe period=768μs) that we used. Upon detection, Contra routed around the failure and recovered the throughput within 1 ms. We found that Hula performs similarly as Contra (Figure 17 in Appendix D).

6.4 Performance: Arbitrary topologies

We now evaluate the performance of Contra on general topologies. We modeled our network after the Abilene [1] topology, configured all links to be 40 Gbps, and randomly chose four pairs of senders/receivers. Since Hula is specialized to a Fattree topology and will not work outside of this context, and since ECMP will not load balance when there is only a single shortest path, we have used two other baselines: a) shortest path routing (SP), which simply sends traffic to the shortest paths, and b) SPAIN [35], which precomputes all paths using (static) heuristics that avoid overlap, and then load balances between these paths.



(a) The web search workload (b) The cache workload

Figure 10: Contra outperforms SPAIN in FCT.

Figure 10 shows the FCT for these different systems. A naïve strategy that simply chooses shortest paths performs the worst. Since SPAIN can utilize multipath routing, it outperforms SP by 32.5% on average for the web search workload and 26.9% on for the cache workload. Contra achieves the best performance among the three: it evenly distributes traffic based on path utilization, and reduces FCT relative to SPAIN by 31.3% on average for the web search workload and 13.8% for the cache workload.

6.5 Protocol dynamics

Next, we study the network dynamics of performance-aware routing. The high-level note here is that, if a policy uses `m` as the metric, then paths with better `m` values may not necessarily be shorter and have lower end-to-end latency. The MU policy is a case in point, because 1) a least-utilized path could be a non-shortest path; and 2) compared to a slightly more utilized shortest path, the non-shortest path may also have higher end-to-end delay if its extra propagation delay offsets its lower queueing delay. Finally, when nodes are temporarily out of sync (§5.5), transient loops may arise.

Transient loops. We first quantify the amount of packets forwarded in transient loops. Under the MU policy on the Fattree and Abilene topologies at 60% load, only tiny fractions of traffic (0.024% and 0.021%, respectively) experienced loops. Compared to the shortest paths, these packets traversed 3.15 more hops on average for the Fattree, and 3.09 for Abilene. They also experienced an increase in end-to-end latency by 72.4μs (Fattree) and 65.2μs (Abilene). Our loop detection mechanisms successfully broke these loops.

Non-shortest paths. Packets could also traverse non-shortest paths without experiencing loops. We observed that the fractions of such packets are 16.6% for Fattree and 39.8% for Abilene. These packets also experienced latency increases of 42.3μs (Fattree) and 39.3μs (Abilene). An interesting observation here is that the application performance (in terms of FCT) depends on the tradeoff between propagation delay and queueing delay. In other words, if least-utilized paths happen to have high propagation delays, then the MU policy may not always lead to FCT improvement. We have observed both cases in our experiments with different topologies, path latencies, and number of shortest paths.

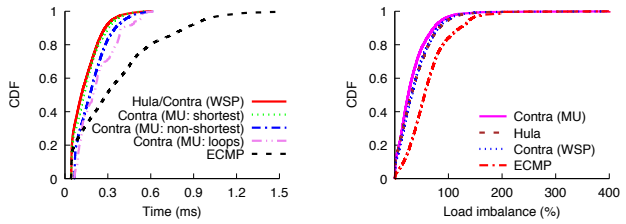
Contra, Hula, ECMP. Figure 11a shows the end-to-end latency packets experienced in Contra for two policies (WSP and MU) on the Fattree, and shows the breakdown for the MU policy (shortest paths, non-shortest paths, and transient loops). As expected, packets never experienced loops under WSP, which routes traffic to the least utilized of shortest paths. For Contra (MU), packets forwarded in loops took more time than those on non-shortest paths, and both took longer than those on shortest paths; they experienced slightly higher latency than in Hula. Nevertheless, Contra (MU) still outperforms ECMP significantly, because shortest paths with high utilization in ECMP have higher queuing delays than non-shortest paths with low utilization. (Results for Abilene are similar; see Figure 19a in Appendix E.)

Network loads. We also found that more packets tend to traverse longer paths as the network load increases (Figures 19b and 19c; in Appendix E). Across all scenarios, 12%-16% traffic took longer paths in the Fattree; within such traffic, 93%-98% took 2 extra hops and the rest took 4 extra hops. On Abilene, where shortest paths are fewer, 32%-50% traffic took longer paths; within this, 97%-99% took fewer than 4 extra hops and the longest path had 9 extra hops.

Load imbalance. Next, we focus on understanding the load imbalance over small timescales in ECMP, Hula, and Contra. Figure 11b shows the CDF of load imbalance of the four aggregate-core links on the Fattree topology at 70% load. We measured the throughput of these links for each 100μs interval for 1 second; we then computed the throughput differences between the most and least loaded links, and normalized them by the average throughput across the links. As we can see, Contra and Hula perform similarly, and they both balance the load much more evenly than ECMP. In particular, this shows that packets traversing non-shortest paths or transient loops in Contra do not lead to notable load imbalance even at small timescales.

6.6 High-fidelity emulation

We have set up a high-fidelity emulator on 15 CloudLab [17] servers using a distributed cluster of Mininet [34]. Our topology is a 6-ary Fattree with 45 switches running P4 `bmv2` [5] and 57 end-hosts with 1:1 oversubscription. We have set the link speeds to be 50 Mbps and verified that this is the highest link speed achievable in our testbed without causing bottle-



(a) End-to-end latency (b) Load imbalance

Figure 11: Packets that experience transient loops or non-shortest paths spent more time in the network (a); but they do not lead to notable load imbalance (b).

necks in the software switches. As discussed before, we have mirrored the same setup in simulation and conducted experiments in both settings. The highest-level takeway is that we observed similar results on both setups. Below, we summarize the key findings, and note that the full set of figures (Figures 20-25) are in Appendix F.

Symmetric Fattree. Contra and Hula outperform ECMP considerably in FCT, and they perform similar to each other. At 90% load for the web search workload, they improve the performance of small flows (<100 kB) by 21%, large flows (>10 MB) by 12.7%, and 13.7% across flows.

Asymmetric Fattree. We then re-ran the FCT experiments, after bringing down three out of 27 aggregate-core links (11.1% reduction to overall capacity). Compared to the topology without failures, at 90% load, the average FCT increased by 15% for the web search dataset and 23% for the cache dataset in ECMP. Contra, on the other hand, only increased by 6% (web search) and 9% (cache), respectively.

Arbitrary topologies. We have also set up the Abilene topology in emulation and compared Contra with SPAIN, and note again that Hula only works on a tree topology. We found that at 90% load, Contra reduces FCT by 22.0% (web search) and 45.7% (cache) under the MU policy. For the web search (cache) dataset, it achieves 11.0% (36.5%) speedup for small flows, and 20.9% (46.1%) speedup for large flows.

7 Related Work

Traffic engineering and load balancing. Contra is different from centralized TE solutions, such as B4 [26], SWAN [25], Hedera [9], MicroTE [15], and Gvozdiev [23], as well as distributed TE solutions, such as TeXCP [28], MATE [18], and Halo [33], in that Contra performs fine-grained load balancing in the data plane. Contra borrows a similar load balancing mechanism from Hula [30] and Conga [11], so their characteristics are similar in terms of dynamics and performance benefits. The main novelty of Contra over all of these systems is to generalize point solutions to a wide range of policies and arbitrary topologies.

Routing. Existing work has studied loop prevention in distance-vector routing [21, 36, 10, 39, 16, 38] with different overhead, convergence, and stability tradeoffs. Contra is most related to DSDV [39], AODV [38], and Babel [16], which use sequence numbers on route updates for convergence. The novelty of Contra lies in its use of programmable data planes to implement a wide array of distance-vector protocols in the presence of unstable metrics, and its design of policy-aware flowlet switching mechanisms.

Regular languages for networking. NetKAT [13], Merlin [45], FatTire [40], and Propane [14] all use regular expressions to specify path constraints, but none of them supports dynamic preferences based on network conditions.

8 Discussion

Correctness guarantees. Contra guarantees that traffic always follows policy-compliant paths even in the presence of unstable metrics. In terms of performance, previous work has shown that when switches make distributed decisions, the resulting mechanism is not globally optimal [11]. However, as demonstrated in our experiments as well as previous work [11, 30], performance-aware routing still leads to FCT improvements compared to static routing.

Policy changes. Policy changes can be handled by recompilation, which would generate new switch programs. As we demonstrated in the experiments, policy compilation is fast and scales to large networks. We expect policy changes to happen infrequently and only on a larger timescale. In order to implement the updates, Contra may be able to borrow existing work on consistent update algorithms [41].

Traffic classes. Contra currently does not support traffic classes. Adding such support would require extending the language with header predicates [20, 13], and designing new mechanisms to prioritize one traffic class over another.

9 Conclusion

We have presented Contra, a system for specifying and enforcing performance-aware routing policies. Policies in Contra are written in a declarative language, and compiled to switch programs that run on the data plane to implement a variant of distance-vector protocols. These programs generate probes to collect path metrics, and dynamically choose the best paths along which to forward traffic. Our evaluation shows that Contra scales well to large topologies, and that the synthesized switch programs can achieve performance competitive with hand-crafted solutions that are specialized to particular topologies and hard-coded policies. Contra is also substantially more general, supporting a wide range of policies over arbitrary topologies.

Acknowledgments: We thank our shepherd Aurojit Panda, the anonymous reviewers, Mina Arashloo, Wei Bai, Shir Landau Feibish, Arpit Gupta, Rob Harrison, Robert MacDavid, and João Sobrinho for their helpful comments and suggestions. This work was supported in part by NSF grants CNS-1703493, FMitF-1837030, and CNS-1801884.

References

[1] Abilene network. https://web.archive.org/web/20120324103518/http://www.internet2.edu/pubs/200502-IS-AN.pdf.

[2] Barefoot Tofino Switch. https://www.barefootnetworks.com/technology/#tofino.

[3] Contra prototype repository. https://github.com/alex1230608/contra.

[4] NS-3 simulator. https://www.nsnam.org/.

[5] P4 behavioral model. https://github.com/p4lang/behavioral-model.

[6] The F# Functional Programming Language. http://fsharp.org/.

[7] The Internet Topology Zoo. http://www.topology-zoo.org/.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. SIGCOMM*, 2008.

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

[10] B. Albrightson, J. Garcia-Luna-Aceves, and J. Boyle. EIGRP – a fast routing protocol based on distance vectors. In *Proc. Networld/Interop*, 1994.

[11] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proc. SIGCOMM*, 2014.

[12] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. SIGCOMM*, 2010.

[13] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *Proc. POPL*, 2014.

[14] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proc. SIGCOMM*, 2016.

[15] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine grained traffic engineering for data centers. In *Proc. CoNEXT*, 2011.

[16] J. Chroboczek. The Babel routing protocol. RFC 6126.

[17] CloudLab. https://www.cloudlab.us/.

[18] A. Elwalid, C. Jin, S. Low, and I. Widjaja. MATE: MPLS adaptive traffic engineering. In *Proc. INFOCOM*, 2001.

[19] B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *Proc. INFOCOM*, 2000.

[20] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *Proc. ICFP*, 2011.

[21] J. J. Garcia-Lunes-Aceves. Loop-free routing using diffusing computations. *IEEE/ACM Transactions on Networking*, 1(1), 1993.

[22] T. G. Griffin and J. L. Sobrinho. Metarouting. In *Proc. SIGCOMM*, 2005.

[23] N. Gvozdiev, S. Vissicchio, B. Karp, and M. Handley. On low-latency-capable topologies, and their impact on the design of intra-domain routing. In *Proc. SIGCOMM*, 2018.

[24] C. Hedrick. Routing Information Protocol. RFC 1058.

[25] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *Proc. SIGCOMM*, 2013.

[26] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proc. SIGCOMM*, 2013.

[27] U. Javed, M. Suchara, J. He, and J. Rexford. Multipath protocol for delay-sensitive traffic. In *Proc. COMSNETS*, 2009.

[28] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *Proc. SIGCOMM*, 2005.

[29] G. Karakostas. Faster approximation schemes for fractional multicommodity flow problems. *ACM Trans. Algorithms*, 4(1):13:1–13:17, Mar. 2008.

[30] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. HULA: Scalable load balancing using programmable data planes. In *Proc. SOSR*, 2016.

[31] P. Kumar, C. Yu, Y. Yuan, N. Foster, R. Kleinberg, and R. Soulé. YATES: Rapid prototyping for traffic engineering systems. In *Proc. SOSR*, 2018.

[32] Q. Ma and P. Steenkiste. Quality-of-service routing for traffic with performance guarantees. In *Proc. IFIP Workshop on Quality of Service*, 1997.

[33] N. Michael and A. Tang. Halo: Hop-by-hop adaptive link-state optimal routing. *IEEE/ACM Transactions on Networking*, 23(6), 2014.

[34] Mininet. `http://mininet.org/`.

[35] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: COTS data-center Ethernet for multipathing over arbitrary topologies. In *Proc. NSDI*, 2010.

[36] S. Murthy and J.J.Garcia-Luna-Aceves. A loop-free routing protocol for large-scale internets using distance vectors. *Computer Communications*, 21(2), 1998.

[37] D. Pei, X. Zhao, D. Massey, and L. Zhang. A study of BGP path vector route looping behavior. In *Proc. ICDCS*, 2004.

[38] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561.

[39] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *Proc. SIGCOMM*, 1994.

[40] M. Reitblatt, M. Canini, A. Guha, and N. Foster. FatTire: Declarative fault tolerance for software-defined networks. In *Proc. HotSDN*, 2013.

[41] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proc. SIGCOMM*, 2012.

[42] J. L. Rodgers and W. A. Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42(1):59–66, 1988.

[43] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proc. SIGCOMM*, 2015.

[44] S. Sinha, S. Kandula, and D. Katabi. Harnessing TCP's burstiness with flowlet switching. In *Proc. HotNets*, 2004.

[45] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *Proc. CoNEXT*, 2014.

[46] Y. Wang and Z. Wang. Explicit routing algorithms for internet traffic engineering. In *Proc. ICCCN*, 1999.

[47] M. Zhang, B. Liu, and B. Zhang. Multi-commodity flow traffic engineering with hybrid MPLS/OSPF routing. In *Proc. GLOBECOM*, 2009.

A Policy analysis and decomposition

At a high level, the Contra compiler implements the policies in a distance-vector protocol, where switches propagate periodic probes and compute a best next hop for each destination using the path metrics. To avoid flooding the network with probes, a switch will only disseminate the best probe in a batch and discard the rest. Moreover, if a policy uses multiple metrics, each probe will carry all metrics to further reduce traffic. However, these techniques are not always safe—the policy needs to be *isotonic*, because otherwise downstream switches can wind up with suboptimal paths. The policy also needs to be *monotonic*, because otherwise loops may form.

Monotonicity. A policy f is monotonic iff. extending a path p by an additional link l does not result in a better ranked path, i.e., $f(p) \leq f(p \cdot l)$; f is *strictly* monotonic if $f(p) < f(p \cdot l)$. Strict monotonicity ensures that loops will not form in distance-vector protocols (assuming static metrics that do not change), because a path's rank only degrades as it gets longer [22]. In principle, one could write a policy that is not monotonic, such as `minimize (- path.len)`, but in practice, we are not aware of such policies actually in use. On the other hand, there are practical policies such as `minimize (path.util)` that are not *strictly* monotonic. To ensure safety, the Contra compiler implements a conservative monotonicity analysis and alerts a programmer of a potential error if the policy is non-monotonic. But our compiler accepts non-strict monotonic programs: our probe propagation mechanism associates an "age" with each probe stored in a switch, and break ties by rejecting more recent probes if they have the same value as the currently used metric, because they may have traversed zero-weight cycles.

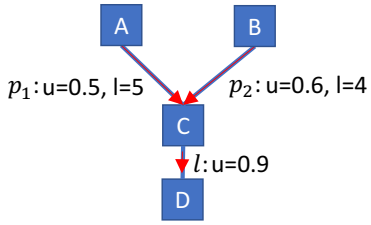


Figure 12: Contra requires (sub)policies to be isotonic.

Isotonicity. A policy f is isotonic iff. for any paths p_1, p_2, and any link l, extending both paths by l preserves the original relative ranking, *i.e.*, $f(p_1) \leq f(p_2) \iff f(p_1 \cdot l) \leq f(p_2 \cdot l)$. Isotonicity guarantees convergence to the best paths [22] even if a switch discards suboptimal probes. Figure 12 demonstrates the idea: if C prefers the probe from path p_1 over that from p_2 and discards the latter, then its downstream neighbor D must have the same preference, or else it would miss a path with a better metric. However, there are some useful policies that are non-isotonic, such as the following congestion-aware routing policy [27] that switches between metrics depending on the network condition.

```
if path.util < .8 then (1, path.util) else (2, path.len)
```

To see why the policy is non-isotonic, consider the switch C in Figure 12 that receives two probes with metrics {`u=0.5,l=5`} and {`u=0.6,l=4`}. C prefers the first probe because `path.util < 0.8` evaluates to true for both probes and the two probes will be ranked based on utilization. However, C cannot simply discard the second probe, because all paths to its downstream neighbor D may be highly congested (e.g., u(D-S)=0.9). In this case, `path.util < 0.8` evaluates to false at D for both probes, causing D's preference to be inverted.

Policy decomposition. The Contra compiler tries to decompose non-isotonic policies into multiple isotonic (and monotonic) subpolicies, and generates different types of probes to propagate each subpolicy. If such a decomposition is impossible, then it rejects the policy. For instance, the compiler decomposes the previous policy as follows:

```
if path.util^0 <.8 then (1, path.util^0) else (2, path.len^1)
```

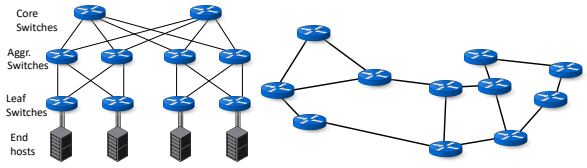
where type-0 probes carry `path.util`, and type-1 probes carry `path.len`. Switches can discard suboptimal probes within each type, but must propagate both types of probes. The complete policy is only evaluated at source nodes. We only attempt this analysis on conditional policies, such as the one above. There remain non-isotonic policies such as "shortest widest paths" (path.util, path.len) that the Contra compiler is unable to implement.

More generally, our compiler performs an analysis to try to decompose f to a collection of subpolicies $(s, f_1, \ldots, f_n)$, where each f_i is monotonic and isotonic, and s combines the subpolicies such that $f(p) = s(f_1(p), \ldots, f_n(p))$. For this decomposition to be correct, s needs to be strictly increasing in each of its arguments, i.e., for any $x_i \leq x_i'$, we need to have $s(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, x_n) \leq s(x_1, \ldots, x_{i-1}, x_i', x_{i+1}, x_n)$. Intuitively, this condition allows a switch to safely discard any non-minimum x_i values of each probe type.

Limitations. Currently, Contra does not support traffic classification, but extending the language with header predicates as in prior work [20, 13] should not present any significant intellectual challenge. A more notable limitation involves policies that prioritize one traffic class over another. For instance, B4 [26] prioritizes small, latency-sensitive user requests over large, latency-insensitive bulk transfers. Currently, Contra ranks paths and selects the best path for each flowlet, but does not compare different types of traffic in order to prefer one over the other. We leave integration of such policies into our framework to future work.

B Key topologies for experimental evaluation

We briefly talked about our experimental setups in Section 6.1. Here, we provide more details of topologies we



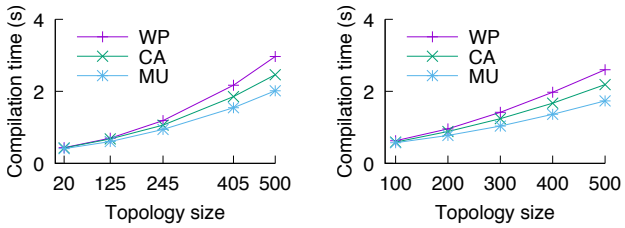
(a) The Fattree topology (b) The Abilene topology

Figure 13: The topologies we have used in the simulation environment.

have used. We conducted our experiments in simulation (ns-3) and emulation (`bmv2` in Mininet) using two topologies—a Fattree network and the Abilene network. Figure 13 shows topologies we used in simulation setup. The links between switches in both topologies operate at high speed: 10 Gbps in Fattree and 40 Gbps in Abilene . The emulation environment, on the other hand, uses a 6-ary Fattree with 45 switches (not shown, see [8] for more details), as well as the Abilene topology (Figure 13b). In both emulated topologies, the links operate at lower speed (50 Mbps) to avoid causing bottlenecks in the software switches.

C Compiler scalability

We tested Contra compiler scalability (more details in Section 6.2) using topologies of varying sizes from 20 to 500 nodes. Figure 14 shows the time to compile P4 programs from high-level policies as topology size increases. Figure 15 shows the amount of resources used by the P4 programs compiled for different policies.

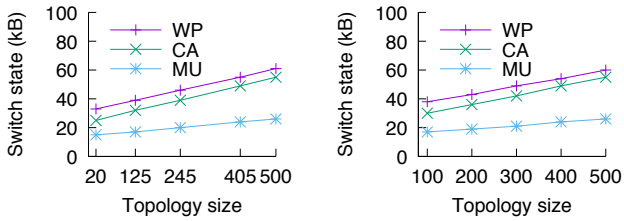


(a) Fattree topologies (b) Random networks

Figure 14: The Contra compiler scales well to large network sizes and sophisticated policies (unit: seconds).

D Link failure

To understand the degree of congestion when topology becomes asymmetric due to link failures, we injected a link failure between aggregation and core switches, and then measured queue lengths of other links. Figure 16 shows the CDF of queue lengths for Contra (under the WSP policy: widest shortest paths) and ECMP when there is a link failure. As we can see, Contra has much shorter queues than ECMP, and Contra queue length is less than 1000 MSS. Note that we have not bounded to queue sizes in order to study the



(a) Fattree topologies (b) Random networks

Figure 15: The Contra compiler generates programs with low memory overhead (unit: kB).

queue growths in different systems. Figure 17 shows the aggregate throughput before and after a link failure. Contra successfully detected this failure in 800 μs and recovered its throughput in 1 ms.

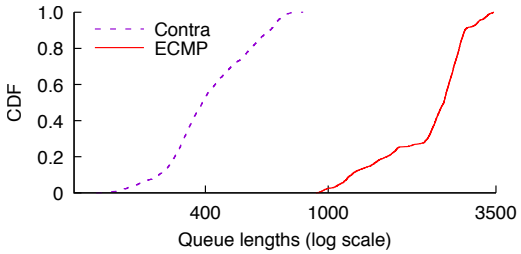


Figure 16: Contra has shorter queues than ECMP.

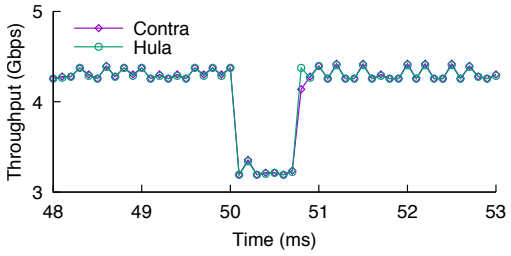
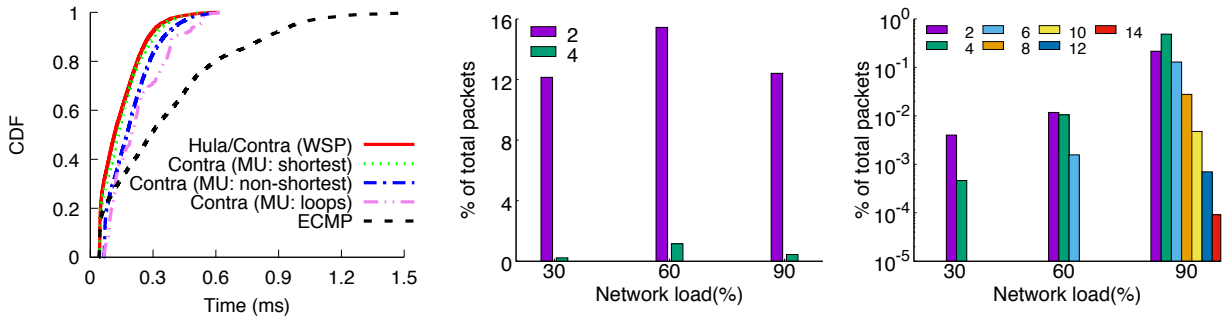


Figure 17: Contra recovers from the link failure within 1 ms.

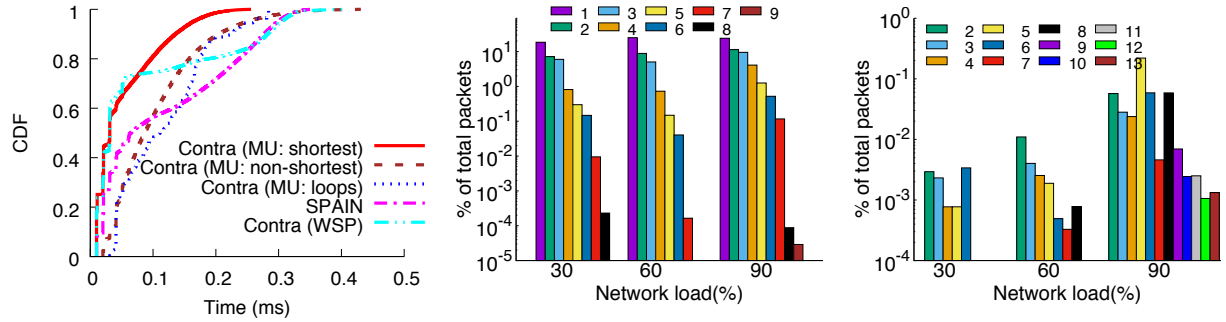
E Protocol dynamics

Previously, in Section 6.5 we have summarized the results of network dynamics experiments. Here, we present the full results across network loads. Figures 18a and 19a show the end-to-end latency packets experience in the Fattree topology and the Abilene topology at 60% network load (dataset: web search). For MU policy, we further break down the number of extra hops in transient loops and non-shortest paths for both topologies: Figures 18b and 19b show number of extra hops in non-shortest paths as network load increases; Figures 18c and 19c show the number of extra hops in transient loops.



(a) End-to-end latency. (b) #Extra hops in non-shortest paths (c) #Extra hops in transient loops

Figure 18: (a): Contra (MU) packets that traverse transient loops and non-shortest paths spent more time in the Fattree network when traffic load is 60%. (b)-(c): breakdowns of extra hops as network load increases; the numbers in the legends denote numbers of extra hops.



(a) End-to-end latency (b) #Extra hops in non-shortest paths (c) #Extra hops in transient loops

Figure 19: (a): Contra (MU) packets that traverse transient loops and non-shortest paths spent more time in the Abilene network when traffic load is 60%. (b)-(c): breakdowns of extra hops as network load increases; the numbers in the legends denote numbers of extra hops.

F High-fidelity emulation

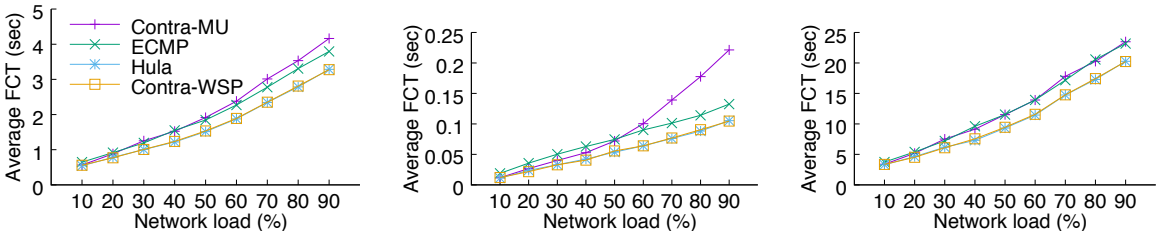
In the main paper, Section 6.6 already summarized the key FCT results obtained in our emulation testbed. Here, we show the full results across workloads, network loads, and Contra policies. All results are obtained over four runs. Figure 20 and Figure 22 show the FCT results for web search and cache workloads in 6-ary symmetric Fattree topology. Figure 21 and Figure 23 show the FCT results for web search and cache workloads in 6-ary asymmetric Fattree topology. Figure 24 and Figure 25 show the FCT results for web search and cache workloads in the Abilene topology.

G Comparison with the FMCF solution

Although the experiments on flow completion times already demonstrate that Contra can boost application performance, we would like to further investigate how Contra performs when compared to an *idealized* solution for which we can derive an optimal bound. To this end, we use a *Fractional Multi-Commodity Flow* problem (FMCF) [29] to model this scenario, and note that similar formulations have been used in other projects [31, 47]. An MCF problem takes as input the (fixed) demand for sender/receiver pairs and the network topology, and computes the optimal traffic splitting across paths in order to minimize the utilization of the most congested link. The *fractional* version of MCF simply means that a flow can be split across different paths as well. This formulation makes several simplifying assumptions, which require minor modifications to the tested systems. Nevertheless, we believe that the results we obtain are still illustrative, as these assumptions make it possible to derive an optimal solution to compare against. The policy we have used in Contra is WSP (widest shortest paths) for Fattree, and MU (Minimum Utilization) for Abilene.

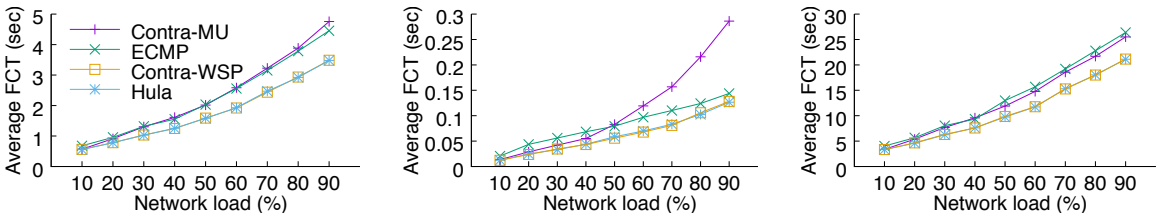
G.1 The FMCF formulation

We have used the same formulation as the Linear Programming Formulation (LPF) in [46]. This formulation models the physical network as $G(V,E)$, where V denotes the set of switches and E denotes the set of links. For each link (i,j), c_{ij} represents its link capacity. $X_{ij}^k \in [0,1]$ is the percentage



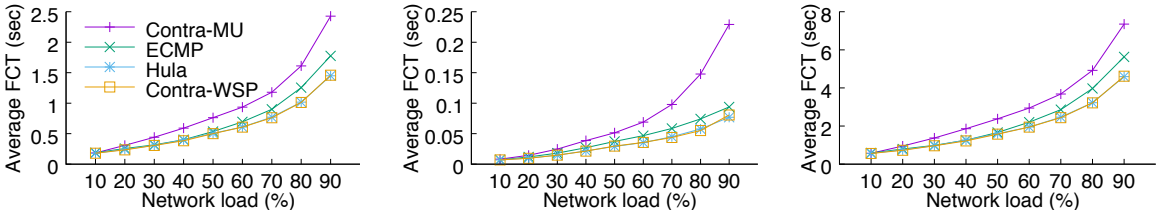
(a) Average FCT of all flows (b) Small flows (<= 100 KB) (c) Large flows (>= 10 MB)

Figure 20: FCT results for web search workload in 6-ary Fattree topology.



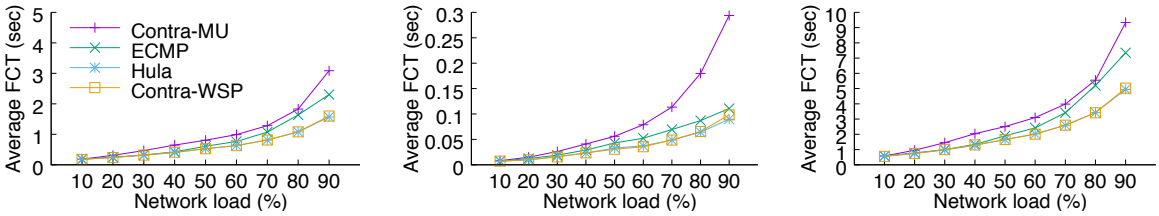
(a) Average FCT of all flows (b) Small flows (<= 100 KB) (c) Large flows (>= 10 MB)

Figure 21: FCT results for web search workload in asymmetric 6-ary Fattree topology.



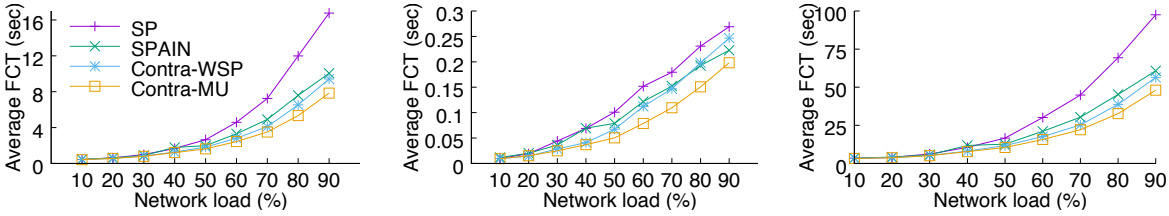
(a) Average FCT of all flows (b) Small flows (<= 100 KB) (c) Large flows (>= 1 MB)

Figure 22: FCT results for cache workload in 6-ary Fattree topology.



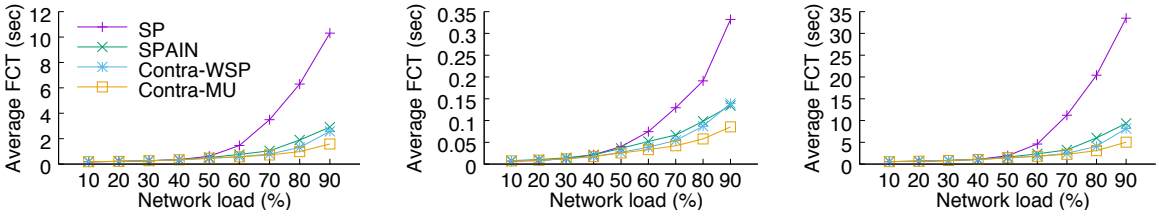
(a) Average FCT of all flows (b) Small flows (<= 100 KB) (c) Large flows (>= 1 MB)

Figure 23: FCT results for cache workload in asymmetric 6-ary Fattree topology.



(a) Average FCT of all flows (b) Small flows (<= 100 KB) (c) Large flows (>= 10 MB)

Figure 24: FCT results for web search workload in Abilene topology.



(a) Average FCT of all flows (b) Small flows (<= 100 KB) (c) Large flows (>= 1 MB)

Figure 25: FCT results for cache workload in Abilene topology.

of traffic a solution sends to link (i, j) for a given commodity flow from the source s_k to the destination t_k, where $k \in K$ represents a commodity flow chosen from the set K of flows to be sent. The total demand for a flow k is d_k. Our goal is to minimize the maximum link utilization $\alpha \in [0, 1]$ across the network.

The problem can then be formulated as follows:

$$\min(\alpha) \tag{1}$$

s.t.

$$\sum_{j:(i,j)\in E} X_{ij}^k - \sum_{j:(j,i)\in E} X_{ji}^k = 0, \qquad k \in K, i \neq s_k, t_k \tag{2}$$

$$\sum_{j:(i,j)\in E} X_{ij}^k - \sum_{j:(j,i)\in E} X_{ji}^k = 1, \qquad k \in K, i = s_k \tag{3}$$

$$\sum_{k\in K} d_k X_{ij}^k \leq c_{ij}\alpha, \qquad (i, j) \in E \tag{4}$$

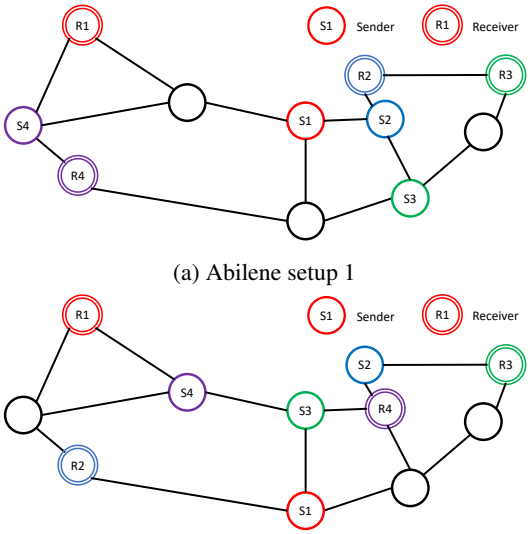
$$0 \leq X_{ij}^k \leq 1, \alpha \geq 0.$$

Equations 1 and 4 define α as the maximum link utilization and set the objective function to minimize this. Equation 2 encodes the flow conservation principle, which specifies that all nodes should have the same amount of incoming and outgoing traffic, except for the sources and destinations. Equation 3 specifies the source switch of each flow.

Simplifying assumptions: We note that this formulation makes several simplifying assumptions when testing the systems. In order to ensure that the created demands are static, we used UDP instead of TCP to avoid its flow control algorithm, and we artificially made the buffers deep enough to avoid packet loss. Given that the FMCF formulation does not have the notion of flowlets (which requires reasoning with timing behaviors), we have configured ECMP, SPAIN, and Contra to perform per-packet load balancing to emulate the problem that FMCF models. This configuration significantly disadvantages Contra, because unlike ECMP, SPAIN, which are inherently multipath, Contra is designed to spread traffic per-flowlet over time, and it only changes paths based on periodic probes. Since this feature is disabled, we instead measured the utilization of all systems at a coarser timescale over multiple RTTs, so that Contra is given an opportunity to balance the load. Despite the above simplifications, we believe that the results we obtain are still illustrative, as these assumptions make it possible to derive an optimal solution that we can compare the actual systems against.

G.2 Experimental results for FMCF

Figures 27 and 28 show three setups where the optimal solutions returned by our solver are 20%, 40%, and 60%, respectively. For each setup, we have tested the systems on a Fattree topology and on Abilene. On a Fattree, both ECMP and Contra are very close to the optimum: they are 0.049% and 0.16% higher than optimum on average. Since ECMP splits traffic on a per-packet basis, it is expected to achieve almost perfect load balancing; Contra underperforms slightly



(a) Abilene setup 1

(b) Abilene setup 2

Figure 26: Selection of sender/receiver pairs on Abilene.

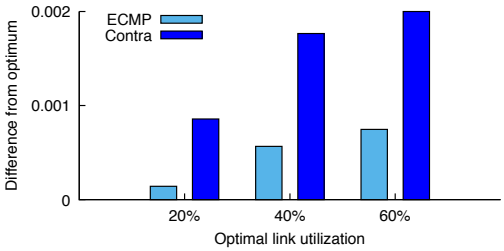


Figure 27: The performance of Contra is close to the optimum in the FMCF formulation (Fattree topology).

since it only changes forwarding decisions based on periodic probes, but it performs close to ECMP and the optimum.

On a general topology, the performance of SPAIN is highly dependent on the locations of senders and receivers, as its load balancing mechanism precomputes non-overlapping paths when possible. We found that, when alternative paths in SPAIN do not overlap, it performs very close to optimum (worse only by 2.53%), but when paths overlap, SPAIN could underperform by as much as 6.55%. Contra, on the other hand, has consistent performance, and achieves similar performance with the same scenarios used to evaluate SPAIN. Compared to the optimum, the results for Contra are 2.41% and 2.36% higher, respectively. Figure 26 shows the two setups we have used for the experiments with SPAIN.

H Waypoint policy

The performance evaluation in our main paper has focused on policies that do not involve regular expressions, because regular expressions constrain paths rather than optimize for performance. Nevertheless, we have conducted a set of ex-

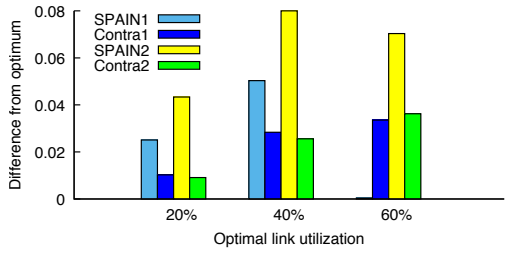


Figure 28: The performance of Contra is close to the optimum in the FMCF formulation (Abilene topology).

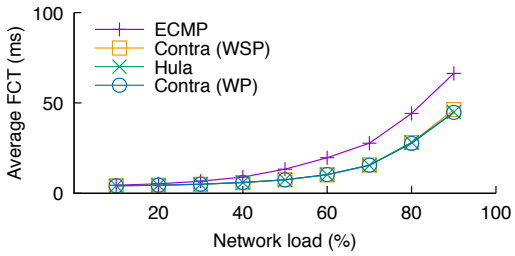


Figure 29: FCT for the waypoint policy (workload: web search)

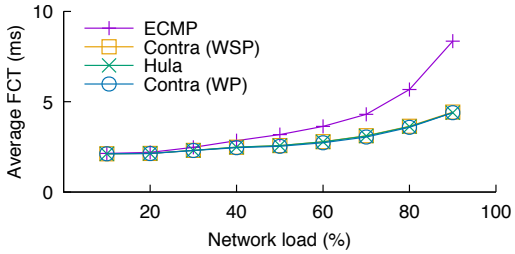


Figure 30: FCT for the waypoint policy (workload: cache)

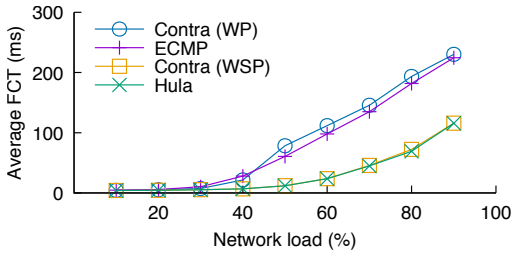


Figure 31: FCT for the waypoint policy on an asymmetric topology (workload: web search)

periments on such policies, and report the key findings in this section. We have used the waypoint policy (WP) with one regular expression, and measured the performance and protocol overhead in different workloads.

Flow completion time. Figures 29 and 30 show the FCT

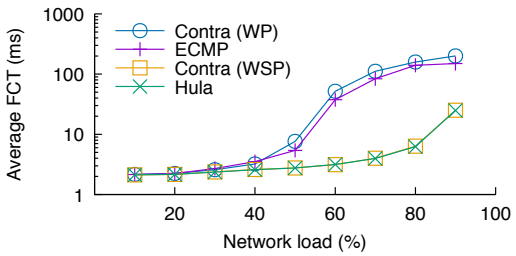


Figure 32: FCT for the waypoint policy on an asymmetric topology (workload: cache)

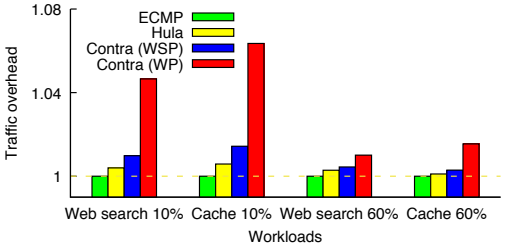


Figure 33: The tags in the waypoint (WP) policy introduce more traffic overhead.

achieved by the WP policy on a symmetric data center topology, on the web search and cache datasets, respectively. As we can see, on the symmetric topology, WP performs similarly to Hula and WSP on both workloads. Figures 31 and 32 show the FCT results for the asymmetric topology, where we have injected a failed link. When the topology is asymmetric, WP performs worse than ECMP. This is expected, as WP imposes additional path constraints.

Protocol overhead. Figure 33 shows the traffic overhead of the WP policy. As we can see, WP sends more traffic than WSP because it tags packets with policy states and creates separate probes for different regular expression matches. At 10% load, 92% of the traffic overhead is due to probes and 8% due to tags; at 60% load, 70% of the traffic overhead is due to probes and 30% due to tags.

I Traffic overhead

To evaluate the traffic overhead incurred by Contra due to additional probes (the policies below do not require tags), we measured the amount of traffic sent over the network by Contra, Hula, and ECMP at 10% and 60% network load. Figure 34 shows the results a normalized by the traffic sent by ECMP (i.e., no extra tags or probes). Across workloads, Contra incurred 0.79% more traffic than ECMP, and 0.44% more than Hula, which seems to be reasonable.

We also evaluated the traffic overhead of SPAIN and Contra on the Abilene network, at 10% and 60% network load. Figure 35 shows the results. We found that only 0.54% of traffic is due to the extra probes in Contra. Interestingly, al-

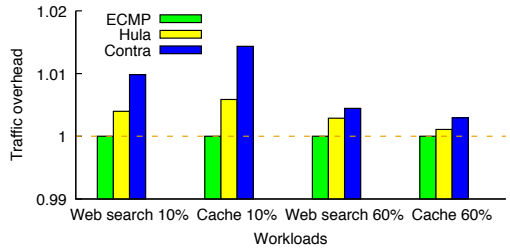


Figure 34: The traffic overhead of Contra is low.

though SPAIN did not use any extra probes, the amount of traffic SPAIN sent across the network is higher than that of Contra, and even higher than the total traffic of Contra. This is because SPAIN's paths are on average longer than these used in Contra. As a result, Contra requires 6.65% less network bandwidth than SPAIN.

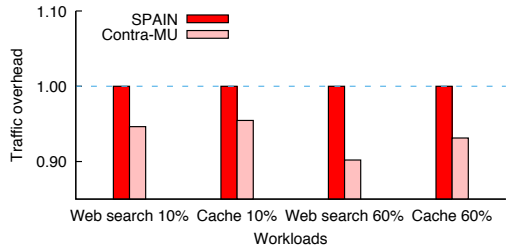


Figure 35: The traffic overhead of Contra and SPAIN on the Abilene network