We introduce a high-level programming language, Contra, which allows users to specify both routing constraints, such as reachability or waypointing, and dynamic optimization goals, such as utility minimization, simultaneously. Each Contra policy denotes a function that maps every path in the network to the relative desirability of that path, which can be based on dynamic information such as the path’s current utilization or latency. The Contra compiler analyzes user policies in conjunction with the network topology, and synthesizes device-local instances of a new distance-vector protocol, implemented in P4. This protocol generates compact probes that gather current path utilities, while avoiding loops, and optimizes for the user’s policy. Switches respond to changing conditions at data-plane speeds by rerouting flowlets according to the current best path for the user’s policy. Our experiments show that our synthesis algorithms scale to large networks, and that in terms of flow completion times, our generated data-plane programs, which may be applied to any topology and a wide range of policies, are competitive with hand-crafted systems that have been specialized to specific topologies and policies.

1 INTRODUCTION
Programming and configuring large networks is a challenging task that network operators face on a day-to-day basis. Networks must satisfy a range of requirements, including security (e.g., do not allow certain hosts to communicate), service chaining (e.g., traffic should go through a series of middleboxes), and traffic engineering objectives (e.g., minimize latency and maximize throughput).

Software-defined networking (SDN) represents a useful step forward, as protocols like OpenFlow and P4 provide powerful mechanisms for achieving multiple objectives at once. Moreover, the programmability of SDN generates new opportunities for innovation and allows flexible customization from one network to the next. As a result, a number of companies have embraced SDN and begun to use it to maximize network performance. However, SDN alone is not a magic bullet. The centralized vantage point of an SDN controller can make networks easier to manage, but it remains inefficient to implement dynamic, fine-grained traffic engineering algorithms with a controller in the loop, as round trip times from switch to controller and back can be too slow to react to changing traffic conditions optimally. As a result, researchers have recently developed algorithms to implement reactive protocols entirely in the data plane [9, 26]. Unfortunately, existing solutions are typically specialized to a single environment. Hence, even if they are provided open source, porting them to networks with different topologies, custom routing policies, or new optimization criteria, can require considerable reengineering and substantial expertise.

A new language for performance-aware routing. To facilitate broad deployment of these responsive, performance-aware routing algorithms, we have developed a new programming language and system called Contra. Contra allows operators to specify custom routing policies (e.g., constraints on the allowed end-to-end paths) and to couple those policies with dynamic performance optimization criteria (e.g., choose the least-utilized paths). The key to our language is a novel programming abstraction that views network policies as total functions that map end-to-end paths to numeric ranks. Whenever a source sends a packet, it is the implementation’s task to use current traffic conditions to choose the path with the best ranking and forward the packet along that path.

The design of our language is directly informed by network programming languages like Frenetic [17], Pyretic [30], FatTire [34], and NetKAT [11], which use (fragments of) regular expressions to classify paths as either legal or illegal. Propane [12] further supports hierarchies of paths—when the first level of the hierarchy is unavailable due to failures, the next level is used. Contra can support both types of policies. Our main linguistic innovation is to extend these static policies with optimization directives on dynamic metrics that depend upon current network conditions. In this respect, it is similar to Merlin [38], which integrates specifications of path constraints and performance. However, Merlin’s programming abstraction, semantics, and implementation are quite different—it computes static rate limits offline, rather than dynamically adjusts to load changes in real time.

Implementation objectives. Our implementation is designed to achieve the following objectives:

1. General – operates over a wide range of metrics
2. Reusable – operates over arbitrary topologies
3. Distributed – does not require central coordination
4. Responsive – adapts to changing metrics quickly
5. Implementable on today’s programmable data planes
6. Policy-compliant – packets only use allowed paths
7. Loop-freedom – mitigates persistent/transient loops

ABSTRACT
We introduce a high-level programming language, Contra, which allows users to specify both routing constraints, such as reachability or waypointing, and dynamic optimization goals, such as utility minimization, simultaneously. Each Contra policy denotes a function that maps every path in the network to the relative desirability of that path, which can be based on dynamic information such as the path’s current utilization or latency. The Contra compiler analyzes user policies in conjunction with the network topology, and synthesizes device-local instances of a new distance-vector protocol, implemented in P4. This protocol generates compact probes that gather current path utilities, while avoiding loops, and optimizes for the user’s policy. Switches respond to changing conditions at data-plane speeds by rerouting flowlets according to the current best path for the user’s policy. Our experiments show that our synthesis algorithms scale to large networks, and that in terms of flow completion times, our generated data-plane programs, which may be applied to any topology and a wide range of policies, are competitive with hand-crafted systems that have been specialized to specific topologies and policies.
Then, Sections 3–5 describe our language and compilation.

To achieve these objectives, the Contra compiler translates network-wide policies into a distributed collection of device-local programs, customized for any target topology. This offline compilation phase first combines the policy and topology into an intermediate representation (IR) that represents all possible legal paths through the network. This IR is an over-approximation of the paths that will be used at runtime, as failures and dynamic performance metrics will cut down the set of preferred paths. A second pass traverses the IR and generates a P4 program customized for each device.

At runtime, the switch programs run a new distance-vector routing protocol that requires no centralized coordination. Switches monitor policy-compliant paths using periodic probes, and collect performance metrics to rank paths. Each switch then stores the best next hop to reach a given destination for forwarding traffic. Since the programs run in the data plane, switches can react to failures and performance changes quickly. However, a naïve implementation that oscillates between paths can cause transient or even persistent chaos, not just forwarding loops but also illegal paths. We draw inspirations from wireless network routing [2, 5, 33], and design mechanisms that leverage programmable data planes to address this. We also design a policy-aware flowlet switching algorithm [37] to mitigate out-of-order packet delivery while ensuring policy compliance.

Summary. Our key contributions follow:

- 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 develop compilation algorithms that generate device-local P4 programs from network-wide policies, satisfying objectives (1)-(10).
- We have built a prototype compiler and runtime, and conducted a series of experiments to compare the performance of our synthesized protocols to a state-of-the-art, manually-coded protocol that has been customized to a specific policy and topology.

In the next section, we further elaborate on a selected set of challenges that would arise if one attempts to reuse low-level, custom algorithms on different policies and topologies. Then, Sections 3–5 describe our language and compilation algorithm, with Figure 1 highlighting how each subsection furthers our objectives. Section 6 evaluates Contra along several dimensions. Section 7 discusses related work, and Section 8 concludes.

## 2 SELECTED CHALLENGES

To understand why it is challenging to achieve the objectives we motivated before, we first describe a strawman solution that only works on a specific topology (data-center networks) and policy (prefer least-utilized paths). We then identify three key challenges that arise in generalizing to arbitrary topologies and more sophisticated policies.

A simple policy on a simple topology. Consider the simple leaf-spine topology shown in Figure 2(a), where switch S wants to send traffic to switch D using the least-utilized path. 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.

More 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 of the inbound link, which is 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 then 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 captured and propagated by the next round of probes.

In fact, this solution describes Hula [26], 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 prevent forwarding loops [26], but on a non-hierarchical topology, loop prevention is much harder.

Consider the sequence of events in Figures 2(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 2(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 gets propagated by D’s periodic probes 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 from now on.

In fact, it is well-known that distance-vector protocols can result in forwarding loops on an arbitrary topology; and over the years, many solutions have been developed [2, 5, 8, 18, 31, 33], each with different tradeoffs. The novelty of our solution is to leverage programmable data planes, where a) switches can add extra header fields to packets, and perform simple, arithmetic operations on header fields, and b) they can maintain a small number of switch-local states [14]. Our solution is inspired by DSDV [33] and a more recent proposal Babel [5], which were originally developed for wireless mesh networks. At a high level, switches assign version numbers to probes, so that they can identify outdated probes and avoid using them. Although transient loops may still occur when probes are in propagation, they will be broken as soon as probes are fully propagated [5].

**Challenge #2: Constrained routing.** Supporting routing policies that admit way-pointing, service-chaining, or other path constraints complicates protocol implementation dramatically. Consider the scenario in Figure 2(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. Under this policy, S can only send traffic to D via a) S-D, b) S-A-D, or c) S-B-D; initially, S prefers c) (u=0.1). Now consider the sequence of events shown in Figures 2(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 (util=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.

In order to address this problem, Contra tags both probes and data packets with policy states that track the paths being traversed, and whether these paths have satisfied the intended policy. When a switch processes a packet, it relies on the embedded tag to determine whether a (local) forwarding decision would or would not violate the (global, network-wide) policy. When a switch changes its path preference, it applies a new tag on packets so that downstream switches know about the change and process the packets based on the latest preference—somewhat akin to a distributed version of consistent updates [35]. This tagging mechanism, again, is made possible by programmable data planes.

**Challenge #3: Custom performance metrics.** Policies with more complex metrics also require a more advanced probe propagation mechanism. In the mechanism we have discussed so far, a switch only propagates the probe with the best metric to its neighbors; this is due to an implicit assumption that probes arriving at a switch with worse metrics can...
be safely discarded, because the metrics will only degrade or, at best, remain the same, when probes are propagated further along a path. However, it is only safe to discard probes when a user’s policy is isotonic [20], which some useful policies are not [23]. Moreover, if policies are non-monotonic, forwarding loops may happen due to negative or zero-weight cycles [20]. To address these problems, Contra performs an analysis to a) discover isotonic subpolicies in a non-isotonic policy and propagate each subpolicy separately, and b) reject certain policies that may jeopardize loop-freedom.

Summary. To summarize, we have designed Contra to support sophisticated, performance-aware routing policies over arbitrary topologies, and to react to network load changes on a data-plane timescale. Figure 1 summarizes the key ideas for Contra to achieve the desirable properties.

3 CONTRA POLICY LANGUAGE

Every Contra policy defines a function that ranks paths. The language has two main components: a) matching on paths using regular expressions, and b) computing path metrics.

3.1 Language by examples

As a concrete example, consider the following policy:

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

It first classifies paths using a regular expression, and then based on the classification, it uses either utilization or latency to select a path. More specifically, traffic that originates at A (i.e., those with paths matching A.*) uses least-utilized paths whereas other traffic is routed based on latency.

Path preferences can also be statically determined from the network topology instead of performance metrics. FatTire [34], for instance, uses regular expressions to classify the legal and illegal paths (though it says nothing about the performance of such paths). For example, to route packets through a waypoint W, a FatTire program would write the policy (. * W .*), which allows any path through W, and no other paths. Such policies can also be represented in Contra by mapping all legal paths to 0 and illegal paths to \( \infty \):

```
minimize( if A . W . then 0 else \( \infty \) )
```

As another example, Propane [12] allows users to write policies such as \((A . * C) \gg (A . * P) \gg (A . * R)\) to indicate relative preferences when sending traffic from A to a customer (C), peer (P), or provider (R). In Contra, we can achieve the same effect by ranking paths statically as below.

```
minimize( if A . C then 0 
else if A . P then 1 
else if A . R then 2 
else \( \infty \) )
```

In such a case, routes to a customer will be used unless a failure makes such routes unavailable, in which case routes through a peer or provider will take over.

When there are many routes to a customer (or peer or provider), one may want to a) take a shortest such path and b) balance load according to current utilization. In Contra, this can be achieved by a lexicographic ordering to rank paths, e.g., by preferring customer over peer over provider first, then shortest paths, and finally, least-utilized paths.

```
minimize( if A . C then (0, path.len, path.util) 
else if A . P then (1, path.len, path.util) 
else if A . R then (2, path.len, path.util) 
else \( \infty \) )
```

Ranking paths using regular expressions defines strict, inviolate preferences; however, operators may have softer constraints as well: 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. To implement soft constraints as well: 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. To implement soft constraints, policies may make different choices based on current path performance. 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 short-utilized paths when network load is heavy (and hence to save bandwidth globally), one might use the following policy.

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

Finally, to steer traffic towards or away from particular links, one may add or subtract weights. For instance, the
Policy: Implementation
T2. Minimum utilization [26] path.util
T3. Widest shortest paths [27] (path.util, path.len)
T4. Shortest widest paths [39] (path.len, path.util)
T5. Waypointing [11] if ‘X’ then path.util else ∞
T6. Link preference [12] if ‘XY’ then path.util else ∞
T7. Weighted link [16] if ‘XY’ then 10 else 0 + path.len
T8. Source-local preference [10] if ‘X’ then path.util else path.lat
T9. Congestion-aware routing [23] if path.util < 0.8 then (1, 0, path.util) else (2, path.len, path.util)

Figure 4: Selected Contra policies.

following policy demonstrates how to add weight to costly links AB and CD while otherwise using simple shortest paths.

\[
\text{minimize}( \text{if } \text{AB } \text{then} \text{else } 0) + \\
(\text{if } \text{CD } \text{then} \text{else } 0) + \\
\text{path.len}
\]

Figure 3 presents the full language syntax, and Figure 4 presents selected examples. The key novelty of the language is that it can capture many of the static conditions expressed by regular languages such as FatTire [34] or NetKAT [11], the relative (static) preferences of Propane [12], and yet combine and augment such policies with dynamic preferences based on current network conditions.

Limitations. Currently, Contra cannot differentiate between flows based on header fields, such as MAC address or TCP port. However, extending the language with header predicates as in prior work [11, 17] should not present any significant intellectual challenge. A more notable limitation involves policies that prioritize one traffic class over another. For instance, B4 [22] 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.

3.2 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 if, 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 [20]. 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.

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 [20] even if a switch discards suboptimal probes. Figure 5 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 [23] that switches between metrics depending on the network condition.

\[
\text{if path.util < 0.8 then } (1, \text{path.util}) \text{ else } (2, \text{path.len})
\]

To see why the policy is non-isotonic, consider the switch \( C \) in Figure 5 that receives two probes with metrics \( (u=0.5, l=5) \) and \( (u=0.6, l=4) \). \( C \) prefers the first probe because \( \text{path.util} < 0.8 \) 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, \( \text{path.util} < 0.8 \) 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:

\[
\text{if } \text{path.util}^{0} < 5 \text{ then } (1 \cdot \text{path.util}^{0}) \text{ else } (2 \cdot \text{path.len}^{1})
\]

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.

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.

### 4 Compilation: Stable Metrics

We describe compilation in two phases. First, in this section, we describe our algorithm as if link utilities do not change, so probes only need to be propagated once. The next section explains how we handle changing metrics.

The most challenging aspect of compilation revolves around policies with conditional regular expression matches such as \((\text{if } r \text{ then } m1 \text{ else } m2)\). If we could determine the path with minimal metric \( m1 \) that matches \( r \), and the path with minimal metric \( m2 \) that does not match \( r \), then we could choose the minimal value among the two to find the best path. However, naively generating a new probe to find the best path for every combination of regular expression matches in the policy would result in a huge number of extra probes for many policies. Instead, the Contra compiler creates a compact data structure that combines all of the regular expression matches together and minimizes the number of generated probes.

#### 4.1 Finding policy-compliant paths

Inspired by Merlin [38] and Propane [12], the first compilation step merges the regular expressions in each policy with the topology to form an intermediate representation (IR) called a **product graph** (PG) that is used to track policy-compliant paths. However, unlike Propane, which computes one PG per destination, we compute a single PG for every destination simultaneously. Construction of the PG assembles the following components:

**Policy automata.** The regular expressions used in a Contra policy define the different ways traffic can flow through the network and affect the path ranking. Because probes disseminate information starting from the destination, they travel in the opposite direction. To capture this, we construct an automaton for the reverse of each regular expression that appears in the policy. Each automaton is a tuple \((\Sigma, Q_i, F_i, q_0, \sigma_i)\). \( \Sigma \) is the alphabet, where each character represents a switch ID in the network. \( Q_i \) is the set of states in automaton \( i \). \( q_0 \) is the initial state. \( F_i \) is the set of accepting / final states. \( \sigma_i : Q_i \times \Sigma \rightarrow Q_i \) is the transition function.

Consider the example policy shown in Figure 6(b), which a) allows A to send traffic to D via the path A-B-D, b) allows B to send traffic to D via any path with the least utilization, and c) disallows all other paths. The Contra compiler would generate the automata shown in Figure 6(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 6(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, then each state in the PG would have \( k + 1 \) fields, \((X, s_1, \ldots, s_k)\), where the first field \( X \) is a topology location, and \( s_j \) is a state in the \( i \)-th automaton; there is a directed edge from \((X, s_1, \ldots, s_k)\) to \((X', s'_1, \ldots, s'_k)\), 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 6(d), every edge represents both valid transitions on the two policy automata and a valid forwarding action on the topology. Notice, for instance, no edges exist from any \((D, *, *)\) state to \((A, *, *)\) state, because such edges have been eliminated due to topology constraints. We use the symbol “−” to denote a special “garbage” state when there is no transition in an automaton. As an example, there is a transition between node D0 and B0 in the PG because a) the topology connects D and B, and b) applying B to each automaton from state 1 leads to state 2.

#### 4.2 Properties of the product graph

Before moving on, we summarize several properties of the PG that we rely on in the rest of the compilation algorithm.

**Virtual nodes.** To distinguish PG nodes from the topology locations, we call the former “virtual nodes” and the latter “physical nodes”. A physical node X may have multiple virtual nodes, because packets could arrive at X via different paths, and as a result, they have satisfied different path constraints and reached different automata states when they arrive at X. 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**. At a high
level, having multiple virtual nodes in the PG means that probes are duplicated in order to find the best path for each path constraint. In the example, node B receives two probes: one for B0 representing a path on the way for regex(ABD), and one for B1 representing a path for regex(B.*D).

**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 $(X_0, \sigma_0(q_0, X), \cdots, \sigma_k(q_k, X))$; all probes that originate from X initially carry this state. This is because, when probes are at the originating node, they have traversed the first character “X” from the initial automata states $q_0$.

**Policy-compliant paths.** Any path through the PG from any state to a probe sending state is a valid, policy-compliant path that is allowed by the policy. All policy-compliant physical paths also exist in the PG.

### 4.3 Packet forwarding

To understand packet forwarding, one needs to understand the structure of the forwarding (FwdT) tables on each switch X. The compiler does not generate the actual forwarding entries for the tables—these are populated at runtime by the protocol logic based on the link metrics, which is described in the following subsection.

Each forwarding table has fields $[\text{dst}^*, \text{tag}^*, \text{pid}^*, \text{mv}, \text{ntag}, \text{nhop}]$, where the fields with stars are used as keys for table lookups. Each row of the table indicates where the given switch will send packets destined for dst when those packets are tagged by product graph node tag and probe number pid. The source of the packets will set the initial tag and the probe number associated with the best path it has found. At each intermediate hop, when a packet with a given dst, tag, and pid matches a particular entry, the switch will look up the next tag (ntag) to write into the packet to replace the current tag, and it will look up the next hop (nhop) to forward the packet to. The metrics vector (mv) is not used during packet forwarding, but it will be used when table entries are populated (following subsection). A property of the forwarding table is that any tag-ntag pair found in a row of a table corresponds to an edge in the product graph, and when a particular 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 product graph—in other words, independently of any other mechanisms in play, forwarding is guaranteed to be policy compliant.

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 regex(ABD), or through b) the best of B-D, B-C-D, and B-A-C-D, satisfying the regular expression regex(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 6(e). The entry is generated from the virtual node B0: if a packet arrives at B with tag=B0 and a destination D, then either that packet was originated at A, and traveled to B or it originated at B. In either case, that tag indicates where in the regex matching process a packet is. In either case, the current best path is through the next hop nhop=D and it has 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, which is D0. The second entry in B’s table is generated from B1. When
packets are tagged with B1, there are two paths they can 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 in this case will be C0. For this policy, the static analysis has determined that only one kind of probe is needed (carrying utilization), so there is only a single probe id of 0. The asterisk next to the entry for B1 indicates that B prefers B-C-D over B-D, which can be determined after evaluating the user policy on both paths (it is easy to evaluate a regular expression match given the PG tag since we know which regexes are accepted in each state). Note that each source can determine its own preference: although B prefers C as the next hop, B will still be able to use A-B-D; A’s traffic will be forwarded using the B0 entry instead.

Function SwiForwardPkt in Figure 7 summarizes the logic for packet forwarding. There is one additional detail we did not mention: 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.

Figure 7: Pseudocode for the synthesized per-device programs. Underlined variables are PG states.

4.4 Managing 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, 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 will add 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. However, as discussed earlier, since probe metrics are monotonically increasing, probes will not be propagated endlessly around 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 6(e)). For example, A must decide whether to use the entry for A0 or A1. 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 state (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 our example, which is initialized to a default value 0); and (4) tag denotes the id of the PG node the probe appears at.

Probe dissemination. The ProcessProbe algorithm describes how a switch processes a probe from its neighbor. This algorithm first obtains the product graph node for the neighbor (n). Next, it updates the metrics in the probe based on the port at which the probe arrived. For instance, it computes and stores the maximum of the probe’s current 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 in the table then it updates its forwarding table FwdT with the next hop, next tag, and metrics vector corresponding to this probe. If an update occurs, then we also need to check if this affects the overall
best choice for the switch (i.e., where the asterisk points to). The BestT table records the current best key for that choice. We look 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 $n_k$ and the probe is multicast to all PG neighbors.

5 COMPILATION: UNSTABLE METRICS

In this section, we describe the techniques used to handle changing path metrics. Section 5.1 describes the high-level idea. Sections 5.2 and 5.5 explain how to handle loops, Section 5.3 ensures timely convergence to optimal paths, and Section 5.4 mitigates out-of-order delivery.

5.1 Strawman solution: Periodic probes

Consider using the same solution as described in Section 4, but instead of sending just one probe, sending many, one per time interval. The packet tagging mechanism will prevent deviations from policy, but packets may still follow (policy-compliant and yet) loopy paths. In fact, the example sequence from Section 2, Figure 2(b)-(f) demonstrates exactly how a problem can arise—the example culminates with the forwarding loop S-A-B-S. Notice also that it is technically policy-compliant because any path from S to D is allowed, so our tagging mechanism will not prohibit it.

The key issue is that when switches use old probes to make decisions, loops can form. In Figure 2(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.

5.2 Preventing persistent loops

To prevent loops, we draw on ideas from Babel [5]. Babel avoids loops by a) distinguishing outdated probes from new ones, and b) discarding 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. We note that, 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 [5].

We note that there is a long body of work on loop prevention in distance-vector protocols. Contra’s compilation algorithm can potentially be integrated with different loop prevention techniques, such as the use of a bit vector to record visited nodes, with tradeoffs being made in terms of space overhead and convergence time.

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 number $v$ with a probe of version number $v'$, it needs to check that $v' \geq v$.

5.3 Probe frequency and failures

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 $\times \text{RTT}$, where $\text{RTT}$ is the highest round-trip time between any pair of switches in the network.

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 expires a FwdT entry if a switch has not received probes for this entry for a certain period of time.

Refinement (Expiration). As before, except that a switch expires a forwarding table entry to D with version number $v$, if it receives a probe from D via another path whose version number is significantly higher (defined as: $v' - v \geq k$) where $k$ is a parameter that determines how fast (in terms of RTTs) failures should be discovered.

5.4 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 at the receiver side. One classic approach is flowlet switching [37], 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. Each switch maintains a table of the form \([\text{fid}^*, \text{nhop}, t]\), where \(\text{fid}^*\) is the flowlet ID (from hashing a packet’s five tuple), \(\text{nhop}\) is the temporarily “pinned” next hop, and \(t\) is the timestamp of the last packet in \(\text{fid}^*\). When the next packet in \(\text{fid}^*\) arrives, the switch computes the gap between its timestamp and \(t\); if the gap is small, this packet will use the current \(\text{nhop}\); otherwise, the switch expires this entry and starts a new flowlet.

Perhaps surprisingly, deploying flowlet switching with Contra may result in policy violations. Consider the example in Figure 8(a), where the policy prefers the least utilized of the upper or lower paths, but avoids the “zigzag” path.

\[
\text{if } \text{SCEFD} \land \text{SAEBD} \text{ then } \text{pathutil} \text{ else } \infty
\]

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 traditional flowlet switching is oblivious to any network-wide routing policy. Our solution works by making it policy-aware. Our insight, again, is that packets can carry tags that represent policy constraints, and in order to ensure policy-compliance, switches need to make forwarding decisions based on the tags. Therefore, policy-aware flowlet switching extends the table format to be \([\text{tag}^*, \text{pid}^*, \text{fid}^*, \text{nhop}, t]\), where \(\text{tag}^*\), \(\text{pid}^*\), and \(\text{fid}^*\) 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 travel the upper path, and would use a separate flowlet switching table entry for the upper path to forward it.

5.5 Breaking transient loops

As discussed in Section 5.2, even with versioned probes, transient loops may still occur when probes are in propagation. Figure 8(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, therefore 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 and flushing the flowlet switching entries. Concretely, each switch maintains a loop detection table \([\text{pkt\_hash}^*, \text{maxttl}, \text{minttl}]\), which maps a packet’s CRC hash to the maximum and minimum TTL values seen at this switch. \(\delta=\text{maxttl}-\text{minttl}\) 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, \(\delta\) would continue to grow. Therefore, switch detects a potential loop (with false positives) when its \(\delta\) 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

Our evaluation is designed to answer three high-level questions: a) How well does Contra scale to large networks? b) How competitive is Contra with hand-crafted solutions? and c) How well does Contra work on arbitrary topologies?

6.1 Prototype implementation and setup

Our Contra prototype consists of 6472 lines of F# [6]. It processes policy programs and topology descriptions specified
in XML, and generates device-local P4 programs. In addition to implementing the algorithms described in this paper, it performs a variety of optimizations, such as minimizing the number of tags, minimizing the forwarding table sizes, and reducing the number of bits to represent the tags.

Our experiments were performed on a Dell T1700 computer, with an Intel E3-1240 CPU with 4 cores and 8 hyper-threads at 3.40 GHz, 8 GB RAM, and a 64-bit Ubuntu 16.04 OS. We have validated our prototype both in Mininet [29] and on ns-3; but our main results were obtained from ns-3, because Mininet does not support large topologies very well. We have used two different types of topologies: a) data center networks, and b) random graphs. Our workloads are sampled from traffic distributions obtained from production networks—a web search workload [10], and a cache workload [36].

6.2 Compiler scalability

To test scalability of our compiler, we used topologies of varying sizes from 20 to 500 nodes. We evaluated three different policies on each topology: a) minimum utilization (MU: no regular expressions, single performance metric), b) way-pointing (WP: three regular expressions, single performance metric), and c) congestion-aware routing (CA: no regular expression, non-isotonic policy with two performance metrics; see p9 in Figure 4). Figure 9 presents the results. The central take-away is that the compiler scales roughly linearly in topology size, and completes in seconds on topologies with hundreds of nodes. Use of regular expressions increases product graph size and hence compilation time. In addition, non-isotonic policies add some overhead due to the additional policy analysis.

6.3 Protocol overhead

To evaluate the traffic overhead incurred by Contra due to packet tags and probes, we used ECMP as a baseline as it balances load without incurring any overhead. To do so, we used a Fat-tree topology with two core switches, four aggregation switches, four leaf switches, and 32 hosts in four pods. The links from hosts to leaf switches are 10 Gbps, and all other links are 40 Gbps; so the network bisection is 160 Gbps with an oversubscription ratio of 1:1. All links have a latency of 10 us. In each experiment, we used 16 hosts as senders and 16 other hosts as receivers, where a sender randomly picks a receiver and sends a flow whose size is sampled from two real-world traffic distributions in production networks: one is a web search workload [10], and the other a cache workload [36].

6.4 Application performance

To evaluate application performance, we compare Contra’s flow-completion times (FCTs) against Hula [26] and ECMP. Hula is a state-of-the-art load balancing scheme designed specifically to minimize utilization over Fat-tree topologies. Figures 11b and 11c show the average FCT achieved by Contra, Hula, and ECMP over the two workloads. Both Hula and Contra outperform ECMP substantially, as the latter cannot avoid poorly performing paths. Under a network load of 80%, Contra reduces the FCT by 10.8× and 40.3× on the web search and the cache workloads over ECMP, respectively. Hula outperforms Contra slightly—by 1.01% on average across network loads and datasets, and by 1.03% under 80% network load. The main reason is still the extra probe traffic that Contra sends. Of course, Contra will perform similarly over any topology and a wide range of policies—Hula can only be deployed on a Fat-tree with a minimum utilization policy.

Figure 9: The Contra compiler scales well to large network sizes and sophisticated policies (unit: seconds).

Figure 10: The Contra compiler generates programs with low memory overhead (unit: kB).
6.5 Arbitrary topologies

To evaluate the benefit Contra provides on a general topology over performance-oblivious routing policies, such as shortest-path routing (SP), we used the Abilene [1] topology with 11 switches and 15 links. All links are configured to carry 40 Gbps with 10 us latency. We measured the average FCT using SP and MU with four pairs of randomly chosen senders and receivers using the web search workload. Figure 11d shows that performance-aware routing (i.e., MU) cuts the FCT by 2.39× on average, and 3.19× with 80% network load, over SP. A closer analysis of the traffic traces shows that transient loops appeared occasionally, but Contra detected them. We note that solutions like Hula and Conga only work on tree-based topologies, and only Contra provides the option to use MU as the policy on an arbitrary topology.

7 RELATED WORK


Data-plane load balancing. Recent work on data-plane load-balancing mechanisms, such as Hula [26], Conga [9], and DRILL [19] perform load balancing at a finer granularity and achieve a faster response to network load changes. They are also utilization-aware — an improvement over simpler mechanisms such as ECMP, which splits traffic randomly regardless of network condition. However, these are mostly point solutions that are specialized for a particular data center topology with a hard-coded, simple policy that prefers least-utilized paths. Contra supports a wide range of sophisticated policies, and works over arbitrary topologies.

Routing protocols and route updates. There is a long line of work on distance-vector routing protocols with a variety of loop prevention techniques [2, 5, 8, 18, 31, 33] with different tradeoffs between probe overhead, convergence time, (in)stability, etc. Contra is most related to DSDV [33], AODV [2], and Babel [5], which use sequence numbers on route updates to achieve timely convergence. A similar idea of using version numbers has been used for consistent updates in SDNs [24, 35]. Compared to existing work, the novelty of Contra lies in its use of programmable data planes to handle loops in distance-vector protocols, and its design of policy-aware flowlet switching.

Regular languages for networking. Contra is related to NetKAT [11], Merlin [38], FatTire [34], path queries [32], and Propane [12] in its use of regular expressions for specifying path constraints. Propane [12] can further specify static path preferences. Contra not only supports static preferences, but also dynamic preferences based on network conditions. Contra is also related to Merlin and Propane in its use of product graphs; but instead of generating BGP configurations, Contra synthesizes switch programs that can run on programmable data planes for performance-aware routing.

8 CONCLUSIONS

In this paper, we have presented Contra, a new high-level language for defining network-wide, performance-aware routing policies. Contra’s policies, which are oriented around the abstraction of path-ranking functions, enhance past network programming language designs by allowing users to combine static policy decisions with dynamic, performance-based metrics. We have also presented a collection of algorithms for compiling these policies: when supplied with an arbitrary network topology, Contra can compile a user policy into a distributed collection of device-local P4 programs. Our evaluation demonstrates that our compiler scales well, as it is able to synthesize programs for networks with hundreds of devices in just seconds. Moreover, the generated programs achieve performance competitive with state-of-the-art, hand-crafted solutions that are specialized to simple, fixed topologies and hard-coded policies.
REFERENCES