SwitchLog: A Logic Programming Language for Network Switches

Vaibhav Mehta$^{1[0000-0003-2357-3023]}$, Devon Loehr$^{1[0000-0003-1127-8932]}$, John Sonchack$^{1[0000-0002-9127-161X]}$, and David Walker$^{1[0000-0003-3681-149X]}$

Princeton University, Princeton NJ 08544, USA
vaibhavm@princeton.edu

Abstract. The development of programmable switches such as the Intel Tofino has allowed network designers to implement a wide range of new in-network applications and network control logic. However, current switch programming languages, like P4, operate at a very low level of abstraction. This paper introduces SwitchLog, a new experimental logic programming language designed to lift the level of abstraction at which network programmers operate, while remaining amenable to efficient implementation on programmable switches. SwitchLog is inspired by previous distributed logic programming languages such as NDLog, in which programmers declare a series of facts, each located at a particular switch in the network. Logic programming rules that operate on facts at different locations implicitly generate network communication, and are updated incrementally, as packets pass through a switch. In order to ensure these updates can be implemented efficiently on switch hardware, SwitchLog imposes several restrictions on the way programmers can craft their rules. We demonstrate that SwitchLog can be used to express a variety of networking applications in a mere handful of lines of code.

Keywords: Programmable Networks · Data Plane Programming. · P4 · Logic Programming · Datalog

1 Introduction

Programmable switches allow network operators to customize the packet processing of their network, adding powerful new capabilities such as for monitoring [6] and performance-aware routing [8]. The core of a programmable switch is a reconfigurable ASIC – a processor specialized for packet operations at high and guaranteed throughputs.

The *de facto* standard programming language for reconfigurable ASICs is P4[2]. However, while P4 allows us to program these devices, it does not make it *easy*. P4 can be viewed as an "assembly language" for line-rate switch programming: it provides fine-grained, low-level control over device operations, but there are few abstractions and, to paraphrase Robin Milner, much can "go wrong."

As a result, researchers have begun to investigate the definition of higher-level languages for switch programming such as Domino [15], Sonata [6], MARPLE [14], Path Queries [13], MAFIA [9], Chipmunk [5], μP4 [19], Lyra [4], Lucid [17, 10], and *Π*4 [3]. Each of these languages provides useful new ideas to the design space—the ultimate future switch programming language may well include components from each of them, which is why exploration of a diverse range of ideas is so valuable now. For instance, Domino, Chipmunk and Lyra improve the switch programming experience by providing higher-level abstractions and using program synthesis to generate efficient low-level code. Sonata and Path Queries, among others, provide new abstractions for monitoring network traffic. μP4 develops a framework that makes switch programs more modular and compositional. Lucid adds abstractions for events and event-handling. *Π*4 and Lucid both develop new type systems for detecting user errors in switch programs.

In this paper, we continue to explore the design space of switch programming languages. In particular, we were inspired by Loo's past work on NDLog [12, 11] and VMWare's Differential Datalog [1], logic programming languages designed for software network controllers. Loo observed that many key networking algorithms, such as routing, are essentially table-driven algorithms: based on network traffic or control messages, the algorithm constructs one or more tables. Those tables are pushed to switches in the network data plane, which use them to guide packet-level actions such as routing, forwarding, load balancing, or access control. Such table-driven algorithms are easily and compactly expressed as logic programs. Moreover, the VMWare team observed that many networking control software algorithms are naturally incremental, so developing an incremental Datalog system would deliver both high performance and economy of notation. Finally, when programs are developed in this form, they are also amenable to formal verification [20].

With these ideas in mind, we developed SwitchLog, a new, experimental, incremental logic programming language designed to run on real-world switches, such as the Intel Tofino. By running in the data plane instead of the control plane, SwitchLog implementations of network algorithms can benefit from finer-grained visibility into traffic conditions [15] and orders of magnitude better performance [17]. However, Because SwitchLog is designed for switch hardware, its design must differ from NDLog or Differential Datalog, which both run on general-purpose software platforms. In particular, arbitrary joins (which are common in traditional Datalog languages) are simply too expensive to implement in the network. Evaluating an arbitrary join requires iteration over all tuples in a table. On a switch, iteration over a table requires processing *one packet for every tuple*, because the switch's architecture can only access a single memory address (i.e., tuple) per packet.

To better support switch hardware, the incremental updates in a SwitchLog program are designed to only require a bounded, and in many cases constant, amount of work. To minimize the need for joins, SwitchLog extends conventional Datalog with constructs that let programmers take advantage of the hardware-accelerated lookup tables in the switch. In a SwitchLog program, each relation specifies some fields to be *keys*, and the others *values.* The rest of the program is then structured so that: 1) most facts can be looked up by their key (a constant-time operation in a programmable switch), rather than by iterating over all known facts; 2) facts with identical keys can be merged with user provided value-aggregation functions.

Although SwitchLog is certainly more restrictive than conventional Datalogs, these restrictions make it practical to execute logic programs on a completely new class of hardware: line-rate switches. To aid programmers in writing efficient programs, we provide several simple syntactic guidelines for writing programs that both compile to switch hardware, and do not generate excessive work during execution. We believe that these guidelines strike a good balance between efficiency and expressivity – in practice, we have found that they still allow us to efficiently implement a range of interesting network algorithms.

To demonstrate how SwitchLog may integrate with other network programming languages, we have implemented SwitchLog as a sublanguage within Lucid[17], an imperative switch programming language that compiles to P4. Within a Lucid program, programmers use SwitchLog to write queries that generate tables of information for use in a larger network application. Once such tables are *materialized*, the rest of the Lucid components may act on that information (for instance, by routing packets, performing load balancing, or implementing access control). Conversely, execution of SwitchLog components may be directly triggered by events in the Lucid program, e.g., a Lucid program can inform the SwitchLog sub-program of new facts. This back-and-forth makes it possible to use logic programming abstractions where convenient, and fall back on lower-level imperative constructs otherwise. While the current prototype is integrated with Lucid, we note that the same design could be used to integrate SwitchLog with any other imperative switch programming language, including P4.

In the follow sections, we illustrate the design of SwitchLog by example, comparing and contrasting it with previous declarative networking languages like NDLog in §2, explaining how to compile SwitchLog to raw Lucid in §3, evaluating our system in §4, and concluding in §6.

2 SwitchLog Design

SwitchLog is inspired by other distributed logic programming languages such as Network Datalog (NDLog) [12]. To illustrate the key similarities and differences, we begin our discussion of SwitchLog's design by presenting an implementation of shortest paths routing in NDLog, and then show how the application changes when we move to SwitchLog.

2.1 NDLog

A typical Datalog program consists of a set of *facts*, as well as several *rules* for deriving more facts. Each fact is an element $r(x_1, \ldots, x_n)$ of some predefined relations over data values x_i. Each rule has the form $p_1 :- p_2, p_3 ... p_n$, where the p_i are facts, and may be read logically as "the facts p_2 through p_n together imply p_1". Operationally, if facts p_2 through p_n have been derived then fact p_1 will be

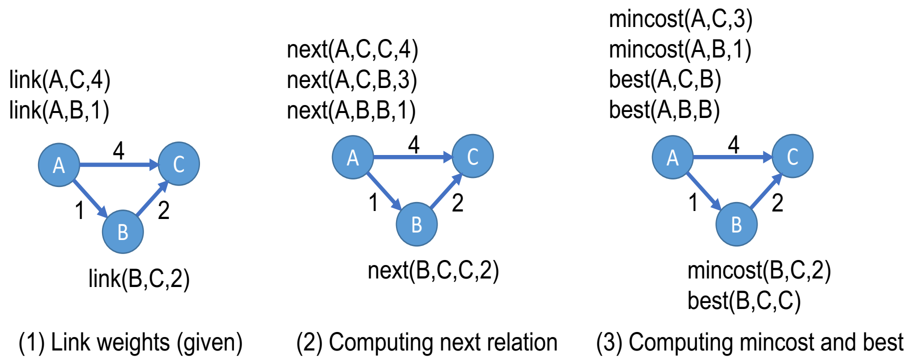

Fig. 1. Sample Execution of NDLog Program

as well. If a relation never appears on the left-hand side of any rule, it is called a *base* relation; elements of base relations are never derived, and must be supplied externally.

In a *distributed* logic programming language like NDLog, each fact is located in a particular place—by convention, the first argument of a fact indicates where it is stored. Hence, the fact $f(@L, x, y)$ is stored at location L—the @ symbol acts as a mnemonic. We call this argument the *location specifier*. Other parameters of the relation may also be locations, but they have no special meaning.

When a rule has the form $f(@L_2, x) :- g(@L_1, y)$, communication occurs: A message is transmitted from location L_1 to location L_2. Hence, such logic programming languages facilitate description of distributed communication protocols (the essence of networking) very concisely.

As an example, consider implementing shortest paths routing in NDLog. We might use the following relations.

```
link(@S, N, C)     // Cost from S to immediate neighbor N is C
next(@S, D, N, C)  // Cost from S to D through N is C
mincost(@S, D, C)  // Cost of min path from S to D is C
best(@S, D, N)     // Best path from S to D goes through N next
```

We will assume the `link` relation has already been defined—for each node S, the `link` relation determines the cost of sending traffic to each neighbor N. To compute `next`, `mincost`, and `best`, we use the following rules.

```
(1) next(@SELF, N, N, C) :- link(@SELF, N, C).
(2) next(@SELF, D, N, C) :- link(@SELF, N, C1),
                            next(@N, D, N', C2), C=C1+C2.
(3) mincost(@SELF, D, min<C>)  :- next(@SELF, D, N, C)
(4) best(@SELF, D, N) :- mincost(@SELF, D, C), next(@SELF, D, N, C)
```

Rules (1) and (2) consider the cost of routing from S to D through every neighbor N. Rule (1) considers the possibility that the destination is the neighbor

N itself — in this case, the cost of the path is the cost of the single link from S to N. Rule (2) considers the cost of routing from S to N (which has cost C1) and then from N to the destination through some other node N' (which has cost C2). The cost of this entire path is C. Rule (3) computes the minimum cost for every source-destination pair and finally Rule (4) finds the neighbor which can reach the destination with the minimum cost.

The key aspect to focus on is the *iteration* involved in running this program. Figure 1 illustrates the results of executing this NDLog program on a particular topology—in each snapshot, the facts generated at node A are shown next to it. The first snapshot presents the link table, which represents the topology of the network from A's perspective. The second snapshot presents the "next" facts that are computed. Snapshot 3 presents the mincost and best relations computed.

To compute `mincost(A, C, 3)` (via rule 3), the logic program considers *all* tuples with the form `next(A, C,` N`,` X`)` and finds the minimum integer X that occurs. In this case, there were just two such tuples (`next(A,C,C,4)` and `next(A, C, B, 3)`), but in more complex topologies or examples, there could be enormous numbers of such tuples. Iteration over large sets of tuples, while theoretically feasible on a switch via recirculation, is too expensive to support in practice.

Likewise, rule (4) implements a *join* between `mincost` and `next` relations. In principle, any pair of tuples `mincost(`S`,`D`,`C`)` and `next(`S`,`D`,_,`C`)` where values S, D, and C might coincide must be considered. Once again, when implementing such a language on switch, we must be careful to control the kinds of *joins* like this that are admitted—many joins demand iteration over all tuples in a relation and/or require complex data structures for efficient implementation that cannot be realized on switch at line rate.

2.2 SwitchLog

As illustrated in the prior section, NDLog programs are very powerful: they can implement arbitrary aggregation operations (like min over a relation) and complex, computationally-expensive joins. Our new language, SwitchLog, may be viewed as a restricted form of NDLog, limited to ensure efficient implementation on a programmable ASIC switch like the Intel Tofino. An essential goal of our design is to ensure that computations never need to do too much work at once; in particular, we must be very careful about which joins we admit, to avoid excessive amounts of iteration.

We achieve this by imagining that our derived facts are stored in a lookup table (a fundamental construct in computer networks), using some of their fields as a key. Then, if the program's rules have the right form (described in §2.3), we are able to look up facts by key, rather than iterating over all facts in the database. This allows us to execute such rules in constant time.

Keys and Values Relations in SwitchLog are declared using the `table` keyword, with the following syntax:

```
// The cost from S to neighbor N is C
table link(@S, loc N : key, int C)
// The best path from S to D is via neighbor N, with cost C
table next(@S, loc D : key, loc N, int C)
```

The type `loc` refers to a location in the network (e.g. a switch identifier). The `key` annotation means that field is a key, so `N` is a key of `link`, and `C` is a value. Similarly, `D` is a key of `next`, and `N` and `C` are values. The location specifier is always implicitly a key. The system maintains the invariant that, for each relation, *there is at most one known fact with a given combination of keys.*

Merging facts To enforce this invariant, every SwitchLog rule contains a *merge* operator, which describes what to do if we derive a fact with the same keys as an existing fact. To illustrate this, consider the following variant of the shortest paths problem in SwitchLog.[1]

```
(S1) next(@SELF, N, N, C) with merge min<C> :- link(@SELF, N, C).
(S2) next(@SELF, D, N, C) with merge min<C> :-
          link(@SELF, N, C1), next(@N, D, C2, hop), C=C1+C2.
```

This program implements the same routing protocol as the earlier NDLog program, but does so in a slightly different way.

Rule (S1) says "if there is a link from `SELF` to `N` with cost `C`, then there is a path from `SELF` to `N` with cost `C`, via `N`." Rule (S2) says "if there is a link from `SELF` to `N` with cost `C1`, and there is a path from `N` to `D` with cost `C2`, then there is a path from `SELF` to `D` with cost `C1+C2`, via `N`." The `min<C>` operator tells us that if we discover another path from `SELF` to `D`, we should retain the one with the lower value of C.

Currently, SwitchLog supports the following merge operators:

1. `min<A>` and retains the fact with the lower value of `A`, where `A` is one of the arguments of the fact on the LHS.
2. `Count<A,n>` increments the argument `A` of the LHS by `n` each time a given fact is rederived. `A` must be an integer, and it must be a value field of the relation.
3. `recent` retains the most recent instance of a fact to be derived (e.g. to simulate changing network conditions).

In the rest of this paper, we may omit the merge operator on rules; doing so indicates that the most recently derived fact is preferred (i.e. we default to the `recent` operator).

[1] A similar program could be implemented in NDLog. It is not that SwitchLog is more efficient than NDLog necessarily, rather that SwitchLog is restricted so that *only* the efficient NDLog programs may be implemented.

The link relation Switches in a network are rarely connected directly to every other switch. Knowledge of how the switches are connected is important for efficient implementation, since sending messages to a distant destination requires more work than communicating with an immediate neighbor.

SwitchLog incorporates this information through use of a special base relation called `link`. The user may determine the exact arguments of the `link` relation, but the first argument (after the location specifier) must be a location. The fact `link(S, N, ...)` means that switch `S` is connected to switch `N`. When this is true, we refer to `S` and `N` as *neighbors*.

2.3 Guidelines for SwitchLog Rules

In addition to providing merge operators, SwitchLog rules must obey several more restrictions to ensure that they can be executed efficiently on a switch. These constraints are detailed below, along with examples. The examples refer to the following relation declarations:

```
table  P1(@S, int Y : key, int Z)
table  P2(@S, int Y : key, int Z)
table  P3(@S, loc Y : key, loc Q : key, int Z, int W)
```

SwitchLog rules must obey the following constraints:

1. No more than two unique location specifiers may appear in the rule.
   `P1(@SELF, Y, Z) :- P2(@A, Y, Z), P2(@SELF, Y, Z)` ✓
   `P1(@SELF, Y, Z) :- P2(@A, Y, Z), P2(@B, Y, Z)` ×

2. If two different location specifiers appear in the rule, then they must be neighbors. (Equivalently, the rule must contain a `link` predicate containing those locations.)

3. If the same relation appears multiple times in the rule, each instance must have a different location specifier.
   `P1(@SELF, X, Y):- P1(@A, X, Y), P2(@SELF, X, Y)` ✓
   `P1(@SELF, X, Z):- P1(@SELF, X, Y), P3(@SELF, X, Y, Q, Z)` ×

   `P1(@SELF, X, Z):- P1(@SELF, X, Y), P3(@SELF, X, Y, Q, Z)`
4. All fields of the LHS must appear on the RHS

5. All predicates on the RHS must have the same set of keys, except that one predicate may have a single additional key, which must be a location.
   `P1(@SELF, Y, Z):- P2(@SELF, Y, Z), P3(@SELF, O, Y, Z, W)` ✓

   `|P1(@SELF, Y, Z):- P2(@SELF,Y, Z), P3(@SELF, O, Y, Z, W)|`
   ✓

   `P1(@SELF, Y, Z) :- P2(@SELF, Y, Z), P3(@SELF, O, Q, Z, W)|`

```
P2(@3, 5, 3)
P1(@SELF, Y, Z) :- P2(@SELF, Y, Z), P3(@SELF, O, Q, Z, W) ×
```

All of these restrictions are syntactic and can be checked automatically. Although in principle, a SwitchLog program that violates these restrictions can be compiled, the compiler we have implemented throws an error for restrictions 1, 3, 4, and 5. A program violating restriction 2 only generates a warning, but still compiles, even though the behavior is not well-defined.

Choosing good guidelines None of the restrictions above are truly fundamental; in principle, one could compile a program that broke all of them, albeit at the cost of significant extra work. We chose these particular constraints because, in our judgement, they strike a good balance between expressivity and efficiency. We show in our evaluation (§4) that we have been able to implement a range of networking algorithms, most of which can execute each rule with a constant amount of work. At the same time, we were still able to implement a routing protocol in which additional work is unavoidable (due to the need to synchronize routing information when a switch is added to the network).

Each of our guidelines reflects some constraint of programmable switches, as well as the fact that switch computation is inherently incremental, dealing with only one packet at a time. These will be described in more detail in the next section, but briefly, the reasons for each rule are as follows:

1. Each different location requires a separate query to look up the fact, and queries from different locations are difficult to merge.
2. If queries are sent between switches which are not neighbors, they must be routed like traffic packets, generating additional work.
3. Switch hardware only allows us to access a single fact of each relation per packet, so we cannot lookup the same fact multiple times in a single location without doing extra work.
4. The predicate on the LHS must be fully defined by the facts on the right; i.e. all of its entries must be "bound" on the RHS.
5. If all the predicates on the RHS have the same set of keys, then given any one fact we can use its keys to look up the values of the others. If a predicate has an additional key, then we need to try all possible values of that key. However, since that key has location type, we need only iterate over the number of switches in the network, rather than a table.

3 Compiling SwitchLog

SwitchLog is compiled to Lucid, a high-level language for stateful, distributed data-plane programming that is itself compiled to P4_16 for the Intel Tofino. Lucid provides an event-driven abstraction for structuring applications and coordinating control that makes it easy to express the high-level ideas of SwitchLog.

A Lucid program consists of definitions for one or more *events*, each of which carries user-specified data. An event might represent an incoming traffic packet

```
// Values for the foo and bar relation (1024 entries each)
global Array.t foo_0 = Array.create(1024); // Stores FOO.v1
global Array.t bar_0 = Array.create(1024); // Stores BAR.v2

// Values for the ABC relation
global Array.t abc_0 = Array.create(1024); // Stores ABC.v1
global Array.t abc_1 = Array.create(1024); // Stores ABC.v2

// Mapping from neighbor to port
global Array.t<<16>> nid_port = Array.create(COUNT);
```

Fig. 2. Arrays representing the SwitchLog relations

to process, a request to install a firewall entry, or a probe from a neighboring switch, for example. Each event has an associated *handler* that defines an atomic stateful computation to perform when that event occurs. Lucid programs store persistent state using the type `Array.t`, which represents an integer-indexed array.

To demonstrate the compilation process, consider the following simple SwitchLog program:

```
table FOO(@loc, int k1 : key, int v1)
table BAR(@loc, int k1 : key, int v2)
table ABC(@loc, int k1 : key, int v1, int v2)

(R1) ABC(@SELF, k1, v1, v2) with merge min<v2> :-
     FOO(@SELF, k1, v1), BAR(@loc, k1, v2), link(@SELF, loc)
```

Note that (`R1`) obeys the all restrictions on SwitchLog Programs described in §2.3; in particular, `FOO` and `BAR` have the same set of keys, except for the location specifier of `BAR`.

3.1 Compiling Relations

The first step to compiling a SwitchLog program is allocating space to store our derived facts. We represent each relation as a hash table, with the keys and values of the relation used directly as the keys and values of the table. Each switch has a copy of each hash table, containing those facts which are located at that switch.

Since modern switches do not allow wide, many-bit aggregates to be stored in a single array, we store each value separately; hence each hash table is represented by a series of arrays, one for each value in the relation. In addition to storing the set of derived facts, each switch maintains a `nid_port` array which maps neighbors to the port they're connected to. Figure 2 shows the arrays in Lucid.

Each SwitchLog relation is also associated with a Lucid event of the same name, carrying the data which defines an element of that relation. The event's

```
memop store_if_smaller(int x, int y) {
    if (x < y)  { return x; }
    else        { return y; }
}
```

Fig. 3. Memops for the Min Aggregate

handler uses the keys of the relation to update the Arrays defined in Figure 2. Then, since other rules may depend on the new information, we trigger an event to evaluate any dependent rules.

3.2 Evaluating Rules

Evaluating a SwitchLog rule requires us to do two things. The evaluation of the rule is always triggered by the derivation of a new fact which matches a predicate on the RHS of the rule. Hence, the first thing we must do is look up facts which match the remaining RHS predicates. If the starting fact contains all the keys of the predicate (e.g. looking up `FOO` in R1 given `BAR`; note that we always know `SELF`), this is simple; we need only use them to look up the corresponding values, then begin evaluating the body of the rule. This is the behavior of `event_bar` in Figure 4.

However, we may not know all the keys in advance. If we start executing R1 with a new `FOO` fact, we do not know which value of `loc` will result in the minimum `v2`. To account for this, we must send queries to *each* neighbor of `SELF` to get potential values of `v2`. Once those queries return, we can use them to execute the body of the rule. In Figure 4, this behavior is split over three events. `event_foo` begins the process, `event_loop_neighbors` sends requests to each neighbor, and `event_lookup_bar` actually performs the lookup at each neighbor and begins executing R1 with the result.

Once we have looked up all the relevant information, the second step is to create the new fact, store it in memory, and trigger any rules that depend on it. This is done in `rule_R1`.

This example illustrates our general compilation strategy. For each relation, we create (1) arrays to represent it as a hash table, and (2) an event to update those arrays. After each update, we trigger any rules that depend on that relation, either by evaluating them directly (e.g., `event_bar`), or by first gathering any necessary information, such as unknown keys (e.g., `event_foo`). The evaluation of rules may trigger further updates, which may trigger further rules, and so on.

The behavior of our compiled code is summarized in Algorithm 1. It is similar to semi-naive evaluation in Datalog in that when a new fact comes in, we only incrementally compute new facts for rules. However, there are two key differences in how this computation is done. The first is that we have to account for inter-switch communication, so we have to query for certain predicates that may be

```
// Update bar's value and execute rule (R1) at all neighbors
handle event_bar(int k1, int v2) {
    int idx = hash<<16>>(SEED, k1);
    Array.set(bar_0, idx, v2);
    generate_ports(all_neighbors, rule_R1(k1, v2));
}
// Update foo's value and (eventually) execute rule (R1)
handle event_foo(int k1, int v1) {
    int<<16>> idx = hash<<16>>(SEED, k1, k2);
    Array.set(foo_0, idx, v1);
    generate event_loop_neighbors(0, k1);
}

// Request the value of bar from each neighbor
handle event_loop_neighbors(int i, int k1) {
    int<<16>> port = Array.get(nid_port, i);
    generate_port(port, event_lookup_bar(SELF, k1));
    if(i < neighbor_ct) {
        generate event_loop_neighbors(i+1, k1);
    }
}

// Look up the value of bar, then execute rule (R1) at the
// requester's location
handle event_lookup_bar(int requester, int k1) {
    int<<16>> idx = hash<<16>>(SEED, k1);
    int v2 = Array.get(bar_0, idx);
    int<<16>> port = Array.get(nid_port, requester);
    generate_port(port, rule_R1(k1, v2));
}

// We don't need v1 because we can look it up
handle rule_R1(int k1, int v2) {
    int<<16>> idx = hash<<16>>(SEED, SELF, k1);
    int v1 = Array.get(foo_0, idx);
    Array.set(abc_0, idx, v1);
    // Store v2 only if it's smaller than the current entry
    Array.setm(abc_1, idx, store_if_smaller, v2);
}
```

Fig. 4. Lucid Code for evaluating rules and updating tables. SEED is a seed for the hash function

missing, and the continue the evaluation when the query returns. The second is that our key-value semantics ensure that one new fact can generate at most one instance of a predicate p_i on a switch.

Algorithm 1 Evaluation Algorithm

```
Require: A fact $q_n$
  for each predicate $P_i$ dependent on $q_n$ do
      if $P_i$ can be evaluated at the current switch then
          $\Delta P_i$ = Evaluate using $\Delta P_0 ... \Delta P_{i-1}$ and rule $R_i$
          $P_i = P_i \cup \Delta P_i$
          if $P_i$ involves communication then
              Broadcast to all switches that need $P_i$
          end if
      else
          if $P_i$ needs a predicate stored elsewhere then
              Store $P_i$
              Query required switches
          end if
      end if
  end for
```

3.3 Optimizations

Accessing memory Switch hardware places heavy restrictions on memory accesses: only a single index of each array may be accessed by each packet, and then only once per packet. This means that we cannot, for example, read a value from memory, perform some computation, and write back to the same memory using a single packet. The general solution is *packet recirculation* – generating a new packet (or event, in Lucid) that does the write in a second pass through the switch. While general, this approach is inefficient because it doubles the number of packets we must process to evaluate the rule.

Fortunately, the hardware provides a better solution if the amount of computation between the read and the write is very small (say, choosing the minimum of two values). The `memop` construct in Lucid represents the allowed forms of computation as a syntactically-restricted function. The argument `store_if_smaller` to `Array.setm` is a memop (defined in Figure 3) which compares `v2` to the current value in memory, and stores the smaller one. By using memops, we are able to avoid costly recirulation.

Inlining events Often, the compilation process described above produces "dummy" events, which only serve to generate another event, perhaps after performing a small computation of their own. Each generated event results in a new packet we need to process, so we inline the generated events wherever possible. This ensures we are doing the maximum amount of work per packet, and thus minimizes the amount of overhead our rules require.

3.4 Integration with Lucid

While SwitchLog programs can be used independently to generate sets of facts (which could be read from switch memory by network operators or monitoring systems), they are most powerful when used to guide packet forwarding within the switch itself. We enable this by embedding SwitchLog into Lucid. Users write general-purpose Lucid code that can introduce new facts to the SwitchLog sub-program, then read derived facts to guide packet processing decisions. For example, using the `fwd_neighbour` array to lookup values and make decisions based on them. This powerful technique allows users to compute amenable data using SwitchLog's declarative syntax, while also operating on that data with the full expressiveness of Lucid. Figure 5 shows how the next table from the routing example can be used to forward packets.

```
table link(@SELF, int dest : key, int cost)
table next(@SELF, int dest : key, int cost, int neighbor)

rule next(@SELF, dest, cost, dest) :-
                link(@SELF, dest, cost)

rule next(@SELF, dest, cost, next) with merge min<cost>:-
                link(@SELF, neighbor, cost1),
                next(@neighbor, dest, cost2, next),
                int cost = cost1 + cost2;

handle packetin(int dst) {
    // check if a path exists.
    int<<16>> idx = hash<<16>>(SEED, SELF, dst);
    // Lookup the `neighbor` value of the stored fwd table
    int next_hop = Array.get(next.neighbor, idx);
    if (next_hop != 255) {
        // Find the matching port and forward the packet
        int outport = Array.get(nid_port, next_hop);
        generate packetout(outport);
    }

}
```

4 Evaluation

We evaluated SwitchLog by using it to implement four representative data-plane applications. The applications have diverse objectives, illustrating the flexibility of SwitchLog.

```
table link(@SELF, int dest : key, int cost)
table fwd(@SELF, int dest : key, int cost, int neighbor)

rule fwd(@SELF, dest, cost, dest) :-
                link(@SELF, dest, cost)

rule fwd(@SELF, dest, cost, next) with merge min<cost> :-
                link(@SELF, neighbor, cost1),
                fwd(@neighbor, dest, cost2, next),
                int cost = cost1 + cost2;

handle packetin(int dst) {
    // check if a path exists.
    int<<16>> idx = hash<<16>>(SEED, SELF, dst);
    // Lookup the `neighbor` value of the stored fwd table
    int next_hop = Array.get(fwd_neighbor, idx);
    if (next_hop != 255) {
        // Find the matching port and forward the packet
        int outport = Array.get(nid_port, next_hop);
        generate packetout(outport);
    }

}
```

Fig. 5. A SwitchLog program integrated with Lucid. The first few lines express a routing protocol in SwitchLog, and the packetin event uses the fwd table

- **Path computation.** The router and mac learner implement routing algorithms at the core of most modern networks.
- **Monitoring.** The netflow cache implements the data structure at the core of many telemetry systems[16][18][6], a per-flow metric cache, while the host usage query implements a per-host bandwidth measurement query from Marple [14].
- **Security.** Finally, the stateful firewall implements a common security protocol – only allowing packets to enter a local network from the Internet if they belong to connections previously established by internal hosts. The distributed heavy hitter detector, based on [7], identifies heavy hitter flows that split their load across multiple network entry points (for example, DDoS attackers seeking to avoid detection).

Our primary evaluation metrics for SwitchLog were conciseness and efficiency, which we gauged by measuring the lines of code and resource requirements of our programs when compiled to the Intel Tofino. Figure 6 reports the results.

Conciseness. As Figure 6 shows, SwitchLog applications are around 10X shorter than the Lucid programs that they compile to, and over 100X shorter than the resulting P4. A SwitchLog program is much shorter than its Lucid equivalent

	LoC			**Resources**			
Application	SwitchLog	Lucid	P4	Stages	Tables	sALUs	Recirculation
Mac Learner	7	55	630	8	11	3	∝ new host
Router	4	74	801	9	19	6	∝ link up/down
Netflow Cache [16]	3	21	320	3	8	2	-
Stateful Firewall	3	28	410	5	10	1	-
Distr Hvy Hitter [7]	5	44	548	3	6	1	-
Flow Size Query [14]	3	58	1276	4	15	7	-

Fig. 6. Lines of code and resource utilization of SwitchLog applications compiled to Tofino.

because each rule in SwitchLog (a single line) translates into many lines of imperative Lucid code that defines the events necessary to propagate new facts and the handlers/memops necessary to perform the respective updates. The resulting Lucid code is itself 10X smaller than the final P4, simply because Lucid's syntax is itself much more concise than P4.

Resource utilization. As Figure 6 shows, all the SwitchLog programs that we implemented fit within the 12-stage processing pipeline of the Tofino. Each stage of the Tofino's pipeline contains multiple kinds of compute and memory resources, for example ternary match-action tables that select instructions to execute in parallel based on packet header values, ALUs that execute instructions over packet headers, and stateful ALUs (sALUs) that update local memory banks that persist across packets. All SwitchLog programs in used under 10% of all stage-local Tofino resources. Our evaluation programs were most resource intensive with respect to the ternary tables and stateful ALUs – Figure 6 lists the total number of each resource required by the programs.

To get an idea of how well-optimized SwitchLog-generated code is, we hand-optimized the Lucid program generated by SwitchLog for the router application with the goal of reducing the number of stages. We found 2 simple optimizations that reduced the program from 9 stages to 7 stages. First, we saved a stage by deleting event handlers that were no longer needed because they had been inlined. Second, we saved another stage by rewriting a memop on the program's critical path to perform an add operation on a value loaded from memory, which previously took place in a subsequent stage. We were unable to find any other ways to optimize the router at the Lucid level, though it is likely that a lower-level P4 implementation could be further optimized, as the Lucid compiler itself is not optimal.

Another important resource to consider is pipeline processing bandwidth. SwitchLog programs recirculate packets when they generate events, and these packets compete with end-to-end traffic for processing bandwidth. Some amount of recirculation is generally okay – for example the Tofino has a dedicated 100 Gb/s recirculation port and enough processing bandwidth to service that port and all other ports at line rate. As the final column of Figure 6 summarizes, we found that in our SwitchLog programs, packet recirculation only occurred when the network topology changes. Such events are rare, thus in practice we expect that all of these applications would fit within the recirculation budget of switches in real networks. We also note that our event inlining optimization was particularly important for the netflow cache. Without the optimization, the cache required a recirculation for *every packet*, whereas with the optimization, it required no packet recirculation at all.

5 Limitations and Future Work

The key-value semantics limit the kinds of programs that can be expressed in SwitchLog. For instance, consider the following NDLog program to compute all neighbors and costs.

```
(1) paths(@SELF, N, N, C) :- link(@SELF, N, C).
(2) paths(@SELF, D, N, C) :- link(@SELF, N, C1),
                             paths(@N, D, N', C2), C=C1+C2.
```

The *paths* table can contain multiple neighbors, which means that one set of @SELF, D values can have multiple values of N and C. However, SwitchLog's semantics only allow one set of values to be stored for each set of keys. Therefore, it is not possible to keep an up-to date table that requires storing multiple values for a given set of keys.

There are also several data plane applications that cannot be expressed in SwitchLog either. A Bloom Filter is a probabilistic data structure that is used commonly on network switches. It consists of an Array, and typically uses a number of different hash functions that map each element to an array index. The current syntax and semantics of SwitchLog do not support the use of multiple hash functions. However, if we allowed multiple hash functions, a bloom filter might look as follows:

```
(1) arr1(@SELF, H, B) :- data(@SELF, K1), H = hash1(K1); B = 1;
(2) arr2(@SELF, H, B) :- data(@SELF, K1), H = hash2(K1); B = 1;
(3) arr3(@SELF, H, B) :- data(@SELF, K1), H = hash3(K2); B = 1;
(4) bloom_filter(@SELF, H, B) :- data(@SELF, K1),
                                 arr1(@SELF, hash1(K1), 1),
                                 arr2(@SELF, hash2(K1), 1),
                                 arr3(@SELF, hash3(K1), 1).
```

where *data* is some SwitchLog predicate. Since SwitchLog is can be integrated with Lucid, Rule (4) could also be implemented as a Lucid function, that computes the hashes and looks up the three arrays. A future target for SwitchLog is to have probabilistic data structures like a Bloom Filter or Count-Min Sketch built-in. In particular, the count aggregate might be implemented as a count-min sketch internally rather than an actual counter.

Another limitation of the declarative syntax is the inability to modify packet headers. In Lucid, this is achieved through the *exit* event. However, allowing programmers to specify exit events in SwitchLog programs would lead to undefined program behavior. In particular, in case of communication, the exit event needs to be a request generated by the SwitchLog compiler, and in case there are 2 possible exit events, it is unclear which one to execute.

6 Conclusion

SwitchLog brings a new kind of logic programming abstraction to the network data plane, allowing programmers to construct distributed, table-driven programs at a high-level of abstraction. While SwitchLog was inspired by past declarative network programming languages such as NDLog [12], the computational limitations of current switch hardware necessitate a modified design; in particular, SwitchLog's explicit key/value distinction enables constant-time lookup of most facts. By restricting the form of rules, SwitchLog can ensure that only efficient joins are permitted. With these restrictions, SwitchLog allows programmers to express a variety of useful networking applications as concise logic programs, and execute those programs inside a real network.

References

1. Differential datalog. VMWare (2019), see also https://github.com/vmware/differential-datalog
2. Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al.: P4: Programming protocol-independent packet processors. ACM SIGCOMM Computer Communication Review **44**(3), 87–95 (2014)
3. Eichholz, M., Campbell, E.H., Krebs, M., Foster, N., Mezini, M.: Dependently-typed data plane programming. Proc. ACM Program. Lang. **6**(POPL) (jan 2022). https://doi.org/10.1145/3498701, https://doi.org/10.1145/3498701
4. Gao, J., Zhai, E., Liu, H.H., Miao, R., Zhou, Y., Tian, B., Sun, C., Cai, D., Zhang, M., Yu, M.: Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication. p. 435–450. SIGCOMM '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3387514.3405879, https://doi.org/10.1145/3387514.3405879
5. Gao, X., Kim, T., Wong, M.D., Raghunathan, D., Varma, A.K., Kannan, P.G., Sivaraman, A., Narayana, S., Gupta, A.: Switch code generation using program synthesis. In: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication. p. 44–61. SIGCOMM '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3387514.3405852, https://doi.org/10.1145/3387514.3405852

6. Gupta, A., Harrison, R., Canini, M., Feamster, N., Rexford, J., Willinger, W.: Sonata: Query-driven streaming network telemetry. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. p. 357–371. SIGCOMM '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3230543.3230555, https://doi.org/10.1145/3230543.3230555
7. Harrison, R., Cai, Q., Gupta, A., Rexford, J.: Network-wide heavy hitter detection with commodity switches. In: Proceedings of the Symposium on SDN Research. pp. 1–7 (2018)
8. Hsu, K.F., Beckett, R., Chen, A., Rexford, J., Walker, D.: Contra: A programmable system for performance-aware routing. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20). pp. 701–721 (2020)
9. Laffranchini, P., Rodrigues, L.E.T., Canini, M., Krishnamurthy, B.: Measurements as first-class artifacts. In: 2019 IEEE Conference on Computer Communications, INFOCOM 2019, Paris, France, April 29 - May 2, 2019. pp. 415–423. IEEE (2019). https://doi.org/10.1109/INFOCOM.2019.8737383, https://doi.org/10.1109/INFOCOM.2019.8737383
10. Loehr, D., Walker, D.: Safe, modular packet pipeline programming. Proc. ACM Program. Lang. **6**(POPL) (jan 2022). https://doi.org/10.1145/3498699, https://doi.org/10.1145/3498699
11. Loo, B.T.: The design and implementation of declarative networks p. 210 p (Dec 2006), http://digicoll.lib.berkeley.edu/record/139082
12. Loo, B.T., Hellerstein, J.M., Stoica, I., Ramakrishnan, R.: Declarative routing: extensible routing with declarative queries. ACM SIGCOMM Computer Communication Review **35**(4), 289–300 (2005)
13. Narayana, S., Arashloo, M.T., Rexford, J., Walker, D.: Compiling path queries. In: Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation. p. 207–222. NSDI'16, USENIX Association, USA (2016)
14. Narayana, S., Sivaraman, A., Nathan, V., Goyal, P., Arun, V., Alizadeh, M., Jeyakumar, V., Kim, C.: Language-directed hardware design for network performance monitoring. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication. p. 85–98. SIGCOMM '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3098822.3098829, https://doi.org/10.1145/3098822.3098829
15. Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., Licking, S.: Packet transactions: High-level programming for line-rate switches. In: Proceedings of the 2016 ACM SIGCOMM Conference. p. 15–28. SIGCOMM '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2934872.2934900, https://doi.org/10.1145/2934872.2934900
16. Sonchack, J., Aviv, A.J., Keller, E., Smith, J.M.: Turboflow: Information rich flow record generation on commodity switches. In: Proceedings of the Thirteenth EuroSys Conference. pp. 1–16 (2018)
17. Sonchack, J., Loehr, D., Rexford, J., Walker, D.: Lucid: A language for control in the data plane. In: Proceedings of the 2021 ACM SIGCOMM 2021 Conference. pp. 731–747 (2021)
18. Sonchack, J., Michel, O., Aviv, A.J., Keller, E., Smith, J.M.: Scaling hardware accelerated network monitoring to concurrent and dynamic queries with {* Flow}. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). pp. 823–835 (2018)

19. Soni, H., Rifai, M., Kumar, P., Doenges, R., Foster, N.: Composing dataplane programs with μp4. In: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication. p. 329–343. SIGCOMM '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3387514.3405872, https://doi.org/10.1145/3387514.3405872
20. Wang, A., Basu, P., Loo, B.T., Sokolsky, O.: Declarative network verification. In: Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages. p. 61–75. PADL '09, Springer-Verlag, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-540-92995-6_5, https://doi.org/10.1007/978-3-540-92995-6_5