The (Surprising) Computational Power of the SDN Data Plane

Calvin Newport
Computer Science Department
Georgetown University
Washington, DC 20057
Email: cnewport@cs.georgetown.edu

Wenchao Zhou
Computer Science Department
Georgetown University
Washington, DC 20057
Email: wzhou@cs.georgetown.edu

***Abstract*—A software defined network (SDN) separates the centralized control plane from the distributed data plane. This approach simplifies control logic at the cost of a heavy burden on the software-based controller and potential long reaction time to data plane events. One solution to this problem is to distribute control logic to multiple controllers spread across the network. Such a solution, however, requires additional mechanisms to enforce correctness properties (e.g., consistency) among the controllers and it still does not fully eliminate latency, as controller decisions happen in software. In this paper, we explore a novel approach to this problem: configuring the rules used by the data plane switches to allow these switches to effectively handle latency-sensitive network management tasks without the direct intervention of the control plane. We are not suggesting to add distributed control logic capability to the switches, we are instead exploring the feasibility of encoding such logic using the standard forwarding rules already available to these devices. To this end, we formally model a network of SDN switches, and then prove using tools from computability theory that such systems are capable of simulating polynomial space Turing Machines, indicating a surprising amount of computational power.**

I. Introduction

A software defined network (SDN) separates the centralized control plane from the distributed data plane. This approach simplifies control logic, allowing innovative strategies for network design and traffic engineering to be deployed with minimal effort. At the same time, however, this centralization puts heavy burden on the software-based controller: when a network event occurs (e.g., a host moves from one location to another or a piece of hardware fails), the control flow has to make an indirection via the controller, and as a result the reaction time can be orders of magnitude slower than an in-network reaction [17]. This is particularly problematic in application scenarios that require fast response to frequent networks dynamics.

One approach to solving this problem is to distribute the (conceptually) centralized control plane to multiple controllers spread across the network, bringing them closer to the data switches and therefore reducing latency and increasing fault-tolerance. This approach, however, still does not fully eliminate the latency caused by the indirection as the controllers are software-based and not every data switch might be proximate. In addition, this distributed approach requires the introduction of additional mechanisms to enforce correctness properties in the control plane, such as consistency, among the multiple controllers. There exists a growing body of research investigating these consistency mechanisms for distributed control [7, 11, 15, 16, 22]. The details of the proposed solutions vary depending on the level of consistency guarantees required, but almost always, they involve heavy algorithmic machinery such as two-phase commit [7] or Paxos-based consensus [15]. These mechanism add an additional delay to the control plane's response to important network events.

Recently, the community has begun to explore a radically different approach to this problem of slow reaction to important network events—an approach that offloads some latency-sensitive network management tasks to the data plane itself, implementing the tasks using only the existing rule-based infrastructure already implemented in the data switches. For example, new versions of the popular OpenFlow SDN system [18] now support conditional rules whose forwarding behavior depends on the local state of the switch; e.g., if links fail, a predefined alternative forwarding port will be chosen. Recent work [21, 2, 3] shows how to leverage this functionality, along with the standard match/action syntax of OpenFlow-style SDN rules, to implement useful tasks such as route repair, topology snapshot, anycast, and black hole/critical node detection without controller intervention. This *in-network* approach is appealing for two reasons: a) it eliminates the extra round trip the controller, and b) it uses only a hardware-based implementation of forwarding rules, not software-based controller logic, leading to a significant performance boost. Indeed, one can speculatively consider applying it to any number of latency-sensitive but critical network operations, including notably those involving route validity or load balancing [4, 9, 13, 14, 19].

These examples are meant to underscore the potential of in-network solutions for certain problems in SDN management. There exists, however, an important obstacle impeding this research direction: the question of *feasibility*. It is perhaps easy to accept that some simple problems, such as recomputing a path in response to a link failure, can be solved using only existing data switch forwarding rule types. But as we consider more complex problems, it becomes less clear that such intricate logic can be implemented using only the simple

types of rules available to a SDN data switch running a standard system such as OpenFlow.

In other words, it might be nice to solve more problems in-network in this setting, but it is unclear that many useful problems *can* be solved in this manner. In this paper, we investigate this intuition by posing the following key question:

> **Question we study:** What class of problems can be solved in-network using only the standard types of rules available to SDN data switches?

We tackle this question from a *theoretical perspective.* In more detail, we formally model the capabilities of an SDN data plane that runs recent version of OpenFlow protocol (e.g., the model in this paper is inspired by OpenFlow v1.3), then apply the tools of computability theory to investigate the computational power of a distributed system with these capabilities. Our investigation reveals a perhaps surprising answer.

> **Answer we prove:** An SDN data plane using only standard OpenFlow-style rules can be configured to solve any problem (defined with respect to its current routing rules) that can be solved by a polynomial-space[1] deterministic Turing Machine.

In other words, even though the SDN data switches might *seem* too limited to compute many conceivable in-network tasks, our formal results prove that they can (in theory) be configured to solve a class of problems powerful enough to capture most tasks relevant to network management. The remainder of this paper is dedicated to proving this result. To do so, we first formally model the capabilities of the SDN data plane. We then describe in our model, for a given Turing Machine (TM), how to carefully configure a set of control rules for the network switches to enable them to correctly simulate this TM running on a tape that begins the computation containing an encoding of the relevant network routing rules. (See Section III for a high-level overview of the strategies used to implement this simulation and the specifics of the results we prove.)

We emphasize that our goal in this endeavor is to prove the *feasibility* of configuring the SDN data plane to solve complex in-network problems using the *rule infrastructure that exists today*. We do not offer in this paper a practical scheme for accomplishing this goal. That is, the TM simulation at the core of our results establishes the computational power of such systems, but is not by itself a particularly efficient or practical strategy to use in a real network deployment (once you leave the safety of asymptotic notation, TM computation is slow, and our simulation requires several packet deliveries and rule updates for every simulated step). We instead intend our results to motivate researchers to continue pushing forward in the quest to identify practical applications of in-network techniques in the SDN context. Put another way: In current SDN systems, controllers typically only delegate the most basic packet forwarding decisions to the data switches; this paper indicates that they could (when useful) be delegating quite a bit more responsibility.

[1]By which we mean space polynomial, roughly speaking, in the number of rules that can be stored at a single switch.

Road Map. The rest of our paper is organized as follows. In Section II, we describe a formal model of the SDN data plane and capture the assumptions used in our computability results. Section III then provides a summary of what we prove and the types of strategies we use in the proofs. Sections IV and V contain our technical results, and finally, Sections VI and VII contain a discussion of related work and our conclusions, respectively.

II. Model

To formalize the computational power of the SDN data plane we need a formal model that abstracts the basic behaviors of these systems (our model focuses in particular on the basic behaviors of OpenFlow networks). The goal is to maintain enough abstraction to enable useful theoretical analysis while maintaining enough details to keep the result relevant to practice.

A. Network Components

In this paper, we describe an SDN's link topology as a connected undirected graph $G = (V, E)$. The nodes in V correspond to network switches, and the edges in E describe links between switches. Each switch $u \in V$ can send and receive packets from its neighbors defined in E. We assume links are reliable and FIFO with bounded message delay. Let A be a set of *addresses* of computational processes/hardware toward which packets received from outside the network might be routed. In this paper, we focus on solving problems defined with respect to the forwarding rules for these destinations. We use $n = \max\{|V|, |A|\}$ to indicate the network size. Finally, we use w_{max} to describe the maximum length of a packet in bits. To obtain the strongest possible results we assume $w_{max} \in O(\log n)$; i.e., the minimum size that can still contain a unique identifier for a switch or address in A.

B. Switch Behavior

Each switch $u \in V$ maintains a collection of *forwarding rules*. An individual forwarding rule is defined as pair $\langle p, X \rangle$, where p is a *pattern* and X is an *action set* containing one or more *actions* that are executed when a packet is received that matches the rule's pattern. In more detail, we assume that each packet w that arrives at a switch u is compared to the pattern component of each rule $\langle p, X \rangle$ stored at u. If w matches p (a notion we define below), then the actions in X are executed. (If X contains multiple actions we allow the rule to specify an order in which they are executed.)

A pattern p is a string in $\{0, 1, *\}^{w_{max}}$, where * represents a wildcard. We say a packet w *matches* pattern p if all symbols that are *not* $*$ in p are the same in w. For example, $w =$010 matches $p =$**0. The core detail in determining the computational strength the SDN data plane are the allowable actions. In standard SDN hardware, the capabilities of these actions are heavily constrained, so we must be vigilant in

defining the allowable actions so as to capture no more than what is possible in most existing SDN systems.

At a high-level, there are two types of actions in an SDN, those specific for data forwarding operations (e.g., forwarding and discarding packets), and those specific for processing control packets of the type typically sent from the controller (e.g., deleting, adding, and modifying the forwarding rules at the given switch). Below we detail the data forwarding and control action types we allow in our model.

1) Forwarding Actions: The following abstract (some of) the standard data forwarding-related actions available in most SDN systems.

- *Forward Unmodified.* This action forwards the triggering packet w, without modification, to a specified neighbor in the network topology.
- *Forward Modified.* This action instructs the receiving switch to forward a modified version of the triggering packet w to a specified neighbor in the network topology. The notion of "modify" used here is heavily constrained to match the limitations of current SDN hardware. In particular, in our model, packet modification is specified by a *template*, which is a packet t, with some bits left unspecified, as well as a mapping from bit positions in the triggering packet w to unspecified bits in t.
  For example, t might equal "01*" and the mapping might indicate that that the switch replace the "*" with whatever bit is in position 2 of the triggering packet. Notice, this definition of modification *only allows* fixed string substitution—preventing non-trivial computation from being embedded into the modification action.

2) Control Actions: The following abstract (some of) the standard control-related actions available in most SDN systems. In this model, we identify a rule (e.g., for the purposes of deleting or modifying) by its pattern match component. We assume, for the sake of simplifying definitions, that only one rule is allowed for a given pattern match (i.e., that these matches provide a unique identifier).

- *Add Rule.* This action adds a specified rule to the receiving switch's rule set.
- *Delete Rule.* This action deletes a specified rule from the receiving switch's rule set.
- *Modify Rule.* This action modifies a specified rule in the receiving switch's rule set. In particular, as with the *forward modified* action defined previously, provides a template t and mapping to specify the modifications. That is, the specified rule can be modified using only fixed bit substitutions from the trigger packet.
- *Forward Rule.* This action forwards a copy of a specified rule at the receiving switch to a specified destination.

3) Rule Size Constraints: We assume that the specification of a given rule can use no more than $O(\log n)$ bits. This restriction prevents a rule from containing too many actions. It also prevents the malformed case where a single action recursively embeds an unbounded number of behaviors.[2]

C. Switch Addressing and Routing

In the below proofs, we show how to simulate a TM using SDN switches passing special control packets back and forth between specially configured switches. To simplify the presentation of these proofs, we handle the identification of switches and the routing of control packets at a high-level of abstraction. That is, we will say things such as "forward this packet to the *tape switch*" (where the *tape switch* is some predetermined switch used in our simulation) to indicate that the packet in question will be routed toward to that switch by some hardcoded set of switch-to-switch forwarding rules.[3]

D. Controllers and Control Actions

At the core of any SDN's operation is the controller (or multiple controllers) that communicates with the data plane switches over a control plane. We do not explicitly model these controllers or the control plane in this paper as our goal is to understand the computational power of the data plane switches in the absence of these centralized entities.

To unlock the computational power of the data plane in the absence of centralized control, however, we leverage the following key assumption: data switches in our model can send and receive packets among themselves that trigger control actions. Notice, in many SDN systems, only packets arriving over the (often secured) control plane can trigger control actions. We require that data plane packets can also trigger these actions—data plane packets that arrive on normal data channels can be escalated and passed to the control plane if they match a predefined pattern. We investigated the complexity of enabling this behavior in a recent version of OpenFlow, and found that it required changes to only a few lines of code. We fully admit, however, that enabling data plane traffic to trigger control rules creates security concerns that we do not address in this paper, as we focus on the feasibility of in-network computation, not the details of making it practical. At a high-level, we suspect that security measures such as role-based authorization can be adopted to thwart security attacks. In addition, these "control packets" could be reported to the controller as the default action for not matching any of the forwarding rules, thereby serving as evidence for accountability schemes.

III. Overview of Formal Results

In Section IV, we formally define what it means for an SDN system to *implement* a specific configuration of a fixed TM M with bounded tape size. This definition specifies a set of switch rules and control packets that must exist in the system. (That is,

[2]For example, consider a rule $\langle p_1, A_1 \rangle$, where A_1 contains an action saying to add rule $\langle p_2, A_2 \rangle$, where A_2 contains an action saying to add rule $\langle p_2, A_e \rangle$, and so on. In this way, the rule definition can be applied in such a way that a single action recursively embeds an unbounded number of rules.

[3]A subtle requirement in one of our arguments below is that the route used between a pair of switches is fixed and FIFO: that is, when we send packets from switch u to v, they arrive in the order we sent them.

if the system contains these specified rules and packets, then it satisfies our definition of *implements* for the configuration and TM in question.) The section concludes with Theorem IV.1, which proves that if the SDN system implements a given configuration at a given time t, then moving forward from time t it will properly simulate M starting from this configuration.

At a high-level, this simulation requires two predesignated switches to pass control packets back and forth. One switch encodes the current contents of the TM tape using its forwarding rules (call this the *tape switch*), and the other encodes the transition function of the TM, also using its forwarding rules (call this the *transition switch*). Our computation simulation operates by looping the following steps: (1) the *transition switch* sends a packet to the *tape switch* informing it what symbol it is writing and what direction it is moving its read head; (2) the *tape switch*, on receiving this packet, updates its tape encoding to capture this newly written symbol (using the *modify rule* action), and sends the *transition switch* back the contents of the tape in its new location. This packet triggers a rule at the transition switch, corresponding to the proper transition for this state and tape value, that will update this packet properly to allow the system to return back to step 1. We emphasize that we implement this behavior using only the standard forwarding rule types defined in Section II.

Having shown that we can configure SDN switches to simulate TM computation starting from a specific configuration, we must now show that we can simulate computation starting from a configuration that encodes the system's *current forwarding rules* on the TM tape (more precisely, the current forwarding rules for the addresses in the set A we fixed in Section II). It is this final result that will establish our main claim: the SDN data plane is able to solve any problem defined with respect to its current forwarding rules that can be solved by an appropriately bounded TM.

To accomplish this more difficult feat, we formally define in Section V what it means for an SDN system to be *primed* for a given TM. As before, this definition specifies a set of data switch rules and packets that must exist in the system. We then prove in Theorem V.1 that if a system is primed for a given TM, and a specially formatted *spark packet* arrives at some predetermined *initialization switch*, then after a finite amount of time the system will satisfy the definition of *implements* (from Section IV) for a configuration that includes the current forwarding rules for A encoded on the simulated TM's tape. It is important to emphasize that this definition of *primed* makes no assumptions about the content of the rules for A. That is, Theorem V.1 assumes that we configure the system to satisfy the definition of primed, *then* arbitrary changes can be made to the forwarding rules for A, *then* the spark packet arrives.

At a high-level, the definition of primed supports a two-phase initialization triggered by the arrival of the spark packet. In the first phase, the arrival of the spark packet initiates a cascade of control packets that lead the switches in the network to send to the *tape switch* (defined in Section IV) enough information regarding their routing rules, so that the *tape switch* can properly update its local representation of the simulated TM's tape to capture encodings of these rules. In the second phase, a termination detection procedure, based on passing a *status packet* on a cycle consisting of all switches, is used to determine when the *tape switch* has received all needed information. Once this state is detected, the proper packet can be sent to begin the simulated computation. As before, we implement this behavior using only the standard forwarding rule types defined in Section II.

We conclude this overview by discussing the specifics of the bounds we assume on the TM's that we simulate. Let $f(n)$ be the maximum number of rules that can be stored on a single switch. In Section IV, we show how to simulate any $g(n)$-bounded TM—i.e., a TM with a tape of size in $g(n)$—starting from any fixed configuration, for some $g(n) = \Theta(f(n))$. The results in Section V assume that the TM tape is large enough to hold an encoding of the network's forwarding rules for addresses in A. There are $O(n^2)$ such rules in the worst case (n switches times $|A| = O(n)$ destinations), each of which requires $O(\log n)$ space to encode with a constant-size TM tape alphabet. It follows that if we want to simulate a TM capable of processing this full set of rules, we need $f(n) = \Omega(n^2 \log n)$. (Of course, if $|A|$ is small, we can reduce this storage requirement, as, more precisely speaking, $\Omega(n|A| \log n)$ space is sufficient.)

IV. Simulating Turing Machine Computation Starting from a Fixed Configuration

We define a *k-bounded TM* $M = (Q, \delta, \Sigma, \Gamma, q_0, q_t)$, to be a standard deterministic single-tape Turing Machine with a bounded tape of size k. A configuration $C = (w, q, i)$ of a k-bounded TM M is defined by its current tape contents $w = w_1, w_2, ..., w_k$, state q, and read head position $i \in \{1, ..., k\}$.

In this section, we first define what it means for an SDN system to *implement* a configuration C of a k-bounded TM M. We then prove that if the system implements C, it will subsequently simulate the computation of M starting at C.

A. Implementing a Turing Machine Configuration

Fix some configuration $C = (w = w_1, w_2, ..., w_k, q, i)$ of a k-bounded TM M. Our construction below will require $\Theta(k)$ rules at certain switches to simulate a TM of this size. Assume in the following that k is small enough to satisfy this condition.

Our definition of implements requires that we have designated some switch to play the rule of a *tape switch* and the other to play the rule of a *transition switch.* This definition will make use of special control rules and packets. We call these *computation rules* and *computation packets*. In more detail, computation packets are control packets with fixed-length fields $type$, $subtype$, $value$, $state$, and $position$, with $type$ always equal to $computation$, while computation rules are those that pattern match packets with $type = computation$.

We say our system *implements* C at a given point[4] if at this

[4]As a modeling subtlety, we refer to a given *point* instead of a given *time*, because it is possible that multiple events can occur at the same time. We assume that the events in an execution can be given some total order, and a point refers to a location in this order.

point the only computation rules and packets in the system are those described below.

1) Tape Switch Computation Rules: We begin by defining the two types of computation rules required for the tape switch by our definition of *implements*. The first set of rules respond to a request from the transition switch to look up the symbol stored in a specified tape position (these rules effectively encode the current contents of the tape of the simulated TM), while the second set of rules are used to update these values.

Formally, for every $i \in [k]$, the tape switch must contain the following computation rule:

- **Pattern Match:** $type = computation; subtype = lookup; position = i$
- **Action #1: Forward Modified**
  - Destination: transition switch
  - Modifications to Forwarded Packet: $value \leftarrow w_i$

In other words, on receiving a computation packet with $type = lookup$ and $position = i$, the tape switch will send the packet back to the transition switch with the $value$ field now updated to include the value contained in position i on the tape according to configuration C.

In addition, for every $i \in [k]$ and $v \in \Gamma$, the tape switch must also contain the following computation rule:

- **Pattern Match:** $type = computation; subtype = write; position = i; value = v$
- **Action #1: Modify Local Rule**
  - Pattern Match of Rule to be Modified: $type = computation; subtype = lookup; position = i$
  - New Action of this Rule: Forward Modified
  - New Action Details:
    - Destination: transition switch
    - Modifications to Forwarded Packet: $value \leftarrow v$.

In other words, on receiving a computation packet with $type = write$, $position = i$, and $value = v$, the tape switch will modify the $lookup$ rule for this position (see above) so that the value in its *forward modify* action is now v. A packet of this type effectively updates the value the tape switch has encoded in its rules for a given tape position.

2) Transition Switch Computation Rules: We now define the single type of computation rule required for the transition switch by the definition of *implements*. This type of rule specifics *two* actions when a packet is matched: one that looks up the tape value in the read head's new location, and one which updates what is stored by the tape switch for the read head's old location. These behaviors are hardcoded in the transition switch rules and determined by the transition function δ of M. Recall from the definition of a TM that for a given $q \in Q$ and $v \in \Gamma$, $\delta(q, v)$ returns three things: (1) the new state; (2) the new value to write; (3) the direction to move the read head after writing. In the following rule description, we capture these three things with the shorthand $\delta(q, v).state$, $\delta(q, v).value$, $\delta(q, v).dir$, respectively.

Formally, for every $q \in Q, v \in \Gamma$ and $i \in [k]$, the transition switch must contain the following computation rule:

- **Pattern Match:** $type = computation; state = q; value = v; position = i$
- **Action #1: Forward Modified**
  - Destination: tape switch
  - Modifications to Forwarded Packet: $subtype \leftarrow write$; $value \leftarrow v'$ (where $v' = \delta(q, v).value$)
- **Action #2: Forward Modified**
  - Destination: tape switch
  - Modifications to Forwarded Packet: $subtype \leftarrow lookup$; $state \leftarrow q'$ (where $q' = \delta(q, v).state$); $position \leftarrow p + b$ (where $b = 1$ if $i < k$ and $\delta(q, v).dir = R$, $b = -1$ if $i > 1$ and $\delta(q, v).dir = L$, $b = 0$ if $i = 1$ and $\delta(q, v).dir = L$, and $b = 0$ if $i = k$ and $\delta(q, v).dir = R$)

In other words, on receiving a computation packet of $type = computation$, $state = q$, $value = v$, and $position = i$, the first action triggered by this rule is to send a $write$ packet to the tape switch that updates what it has stored in i to reflect what M would write if it was in state q and reading v. The second action then creates a $lookup$ packet with the simulated machine's new read head position and state, according to the definition of its transition function, and sends it back to the tape switch to initiate the next simulated step of computation.

3) Computation Packets: Finally, to satisfy our definition of *implements* for configuration C at a given point, the system must contain exactly one computation packet at this point: a packet being sent to the transition switch with $state = q$, $position = i$, and $value = w_i$.

B. The Simulation Theorem

Now that we have defined what it means for an SDN system to implement a configuration C of a TM M, we now prove that if the system satisfies this definition it will subsequently accurately simulate M starting from C. Our definition of simulate captures the following behavior. Let $C, C_2, C_3, C_4, ...$ define the uniquely defined sequence of configurations that occur when we begin executing deterministic TM M starting from C. Our below theorem shows that once we implement C, after a finite amount of time we implement C_2. We can then of course keep reapplying the theorem to show that once we implement C_2 we eventually get to C_3, then to C_4, and so on. Furthermore, the theorem requires that no computation packets are sent to the transition switch in between these times, allowing us to consider these packets to be an effective clock for the simulation; i.e., each time the tape switch sends a new computation packet we have implemented the next configuration in the TM's computation.

Theorem IV.1. *Assume that at some point t an SDN system implements some configuration C of our fixed TM M. There exists a future point t' such that at t' the SDN system implements configuration C', where C' is the configuration that follows C by the definition of M. Furthermore, in the*

interval between t and t' the tape switch does not send any computation packets.

Due to space limitations, we omit theorem proofs in this paper, and refer interested readers to the technical report [1].

V. Simulating Turing Machine Computation on the Current Forwarding Rules

In the previous section, we described how to configure an SDN system to simulate a bounded TM M starting from a fixed configuration C. In this section, we leverage these results to prove something more powerful: how to configure a system for a fixed M such that on the arrival of a *spark packet*, it will begin simulating M starting from a configuration that has the switch's current relevant routing rules encoded on the tape (where "relevant" in this context means the forwarding rules for some fixed destination address set A).

In more detail, we formally define what it means for the system to be *primed* for a given TM M. We then prove that if the system satisfies this definition, and then subsequently (perhaps after arbitrary updates to the rules for A), a spark packet arrives, the system will update its control rules such that it satisfies the definition of *implements* from Section IV for M in its initial state on a tape containing the current forwarding rules for A (in some fixed encoding). At this point, Theorem IV.1 from Section IV applies to tell us that the system will continue on to correctly simulate M starting from this configuration.

Preliminaries. Before proceeding to the details, we must first establish some helpful preliminary definitions and assumptions. Fix some k-bounded TM M to use in the remainder of this section. Because we want to run this TM on a tape that contains the full set of forwarding rules in the network for destinations in A, we need, in the worst case, for k to be in $O(n^2 \log n)$ (n switches, n destinations, and $\log n$ space to encode each next-hop). As established in Section IV, to simulate computation on a tape of length k requires the ability to store $\Omega(k)$ rules at a given switch. In the following, therefore, we must assume that each switch can hold $\Omega(n^2 \log n)$ rules.

Without loss of generality, in the following we also assume that for each $a \in A$, each switch has a single forwarding rule. The following results easily extend to the more general case where a switch might have multiple rules for a given destination (though at the cost of increased notational clutter).

Finally, we note that in Section IV, the number of different types of control rules included in the relevant definitions were limited. We were able, therefore, to provide technical definitions of each type of rule (i.e., formally define the pattern match and resulting actions). In this section, the definition of *primed* includes many more types of rules. For the sake of intuition and concision, therefore, we use slightly higher-level (and therefore more clear and compact) textual descriptions of these rules. Using the examples from Section IV as guidance, the translation of these descriptions to fully formally definitions should be straightforward and unambiguous.

A. Priming an SDN for a Given Turing Machine

Our definition of *primed* requires we fix in advance some switch to play the rule of the *initialization switch*. The definition also makes use of special control rules and packets that we call *initialization rules* and *initialization packets.* In more detail, we assume initialization packets have fixed-length fields $type$, $subtype$, $target$, and $payload$, with $type = init$, and that initialization rules pattern match packets with $type = init$.

We say our system is *primed for M* at a given point if at this point the only initialization *and* computation rules and packets in the system are the following.

1) Computation Rules and Packets: For the system to be primed for M at a given point, we assume the only computation rules in the system at this point are exactly those required to *implement* M in an initial configuration with an empty tape (see Section IV). Further, we require that at this point there are no computation packets active in the system.

2) Initialization Rules at the Initialization Switch: Our definition of primed requires the initialization switch to contain the initialization rules defined below, which are unique to its role as an initialization switch (we will subsequently define some additional rules required of all switches by the definition of primed, including the initialization switch).

Fix some arbitrary total ordering on V, the set of switches in the SDN system. The initialization switch must have one *trigger rule* for each $u \in V$. The trigger rule for u pattern matches to an initialization packet with $subtype = trigger$ and $target = u$. This rule corresponds to two actions. The first action generates a trigger packet for the next switch in V (if any) by our above fixed total ordering and forwards this packet to itself (i.e., over the loopback interface) to be processed. The second action generates an initialization packet with $subtype = record$ and $target = u$. It forwards this packet to u. We call this latter type of packet a *record packet* for u.

As mentioned, the initialization process is initialized by some distinguished *spark* packet (by "distinguished" we mean that the packet is in a format that is only used for this purpose) arriving at the initialization switch. To handle this event, our definition of primed also requires the initialization switch to have a rule that pattern matches this spark packet. This rule has a single action which has the initialization switch send itself a trigger packet for $u_0 \in V$, where u_0 is the first packet by our fixed total ordering.

3) Record Rules at all Switches: Our definition of primed requires that every $u \in V$ has a rule that pattern matches an initialization packet with $subtype = record$ and $target = u$. This rule has a single action: u sends an initialization packet to itself with $subtype = localrecord$ and $target = b$, where $b \in A$ is the first address in A by some fixed total ordering.

For every $a \in A$, node u also must contain a rule that pattern matches $subtype = localrecord$ and $target = a$. This rule has two actions. The first action has u send itself a packet with $subtype = localrecord$ and $target = a'$, where a' is the next address in A by our ordering (for the last address in this ordering, this action is omitted). The

second action has u forward itself an initialization packet with $subtype = localrecord2$ and including its current routing rule for destination a in the packet's $payload$ field. (This requires the *forward rule* action that we defined in our model as one of the common control actions available in standard SDN switches. Notice, this is where we leverage our simplifying assumption that there is only one such rule for a at u.)

The effect of a $subtype = record$ packet arriving at u is that u will subsequently send itself, for every $a \in A$, a $subtype = localrecord2$ packet containing its routing rule for a in the $payload$ field. Our goal is to add some additional initialization rules to our definition of primed such that each $localrecord2$ packet arriving at u generates a series of computation packets, sent to the tape switch, that will update its representation of the TM tape to include this routing rule in a proper format.

In more detail, for a given packet p arriving at u with $subtype = localrecord2$ and $target = a$, let $p.payload$ be the fixed bits in the packet that correspond to the payload field containing u's forwarding rule for a. The bits in $p.payload$ describe, among other things, a next hop for these packets. Let $p.payload.dest$ be the fixed bits in $p.payload$ that correspond to encoding this next hop . We assume that $p.payload.dest$ is of size $O(\log n)$ bits (i.e., the rules are encoded in a reasonable manner). Finally, we can address single bits in subfield using the notation $p.payload.dest[i]$, for each bit position i. Without loss of generality, we assume in the following that the encoding of routing rules assumed by M simply places the bits from $p.payload.dest$, for each switch and destination, in some fixed block of cells on the TM tape.

Our definition of primed requires two $localrecord2$ rules for each target a and bit position i in $p.payload.dest$. One rule pattern matches $localrecord2$ packets for this target with $p.payload.dest[i] = 0$, and the other matches such packets with $p.payload.dest[i] = 1$. Each of these rules specifies the value to be encoded on the corresponding cell of the simulated TM's tape. Therefore, the action components of these rules must sent the appropriate computation packet to the *tape switch* to update the value it has encoded for that location. (See Section IV for details on the tape switch and computation packets.) This writing step can be accomplished by forwarding a computation packet with $subtype = write$ to the tape switch, with the $position$ and $value$ field set appropriately.

4) Synchronization Rules at all Switches: Our definition of primed so far has required switches to contain rules that will implement the following behavior: when the initialization switch receives a spark packet, all switches will subsequently receive a $record$ packet that will cause each to cycle locally through all destination in A, for each sending the appropriate $write$ computation packets to the *tape switch* to encode the rule in the proper location on the simulate TM's tape. Once every switch has sent these $write$ packets to the tape switch, and the tape switch has processed them, the computation rules in the system are correct for the configuration we want to implement. At this point, all that is required is for the appropriate computation packet to be sent toward the *transition switch* to satisfy our definition of *implements* for this configuration.

It is here, however, that we encounter our final technicality. The message delay in our distributed network is uncertain. It follows that some switches might finish processing their local routing rules faster than others. Though it is not hard to inform a given switch when its own packets have been processed by the tape switch (see below), it is more difficult to determine when *all* switches have learned they are done.

In this section, we define a final set of initialization rules required by our definition of primed that help synchronize these updates to the tape switch so that the initialization switch can detect when all updates are complete. In more detail, we first add a series of rules at every switch that respond to initialization packets with $subtype = poll$. We configure the switches to pass these packets in some fixed cycle. That is, when u receives a $poll$ packet its relevant rule forwards it to the next hop in this cycle. The key to this procedure is that we also make use of a $done$ field in these packet. This field is either set to 0 or 1 in any given $poll$ packet. We configure every switch for two $poll$ packet rules: one that pattern matches this bit equaling 1 and another that pattern matches 0. For every switch *except* the initialization switch, the default state for these rules is that in both cases, it forwards the packet to the next hop in the cycle with $done \leftarrow 0$. The default state for these rules at the initialization switch is to forward the packet to the next hop with $done \leftarrow 1$, regardless of the value of $done$ on its arrival from its predecessor in the cycle.

Next, for each switch, we update the two required $localrecord2$ rules corresponding to the final bit position in $p.payload.dest$ for the final address in a in the fixed total ordering used in our local cascade of $localrecord$ rules at u. In particular, we add a new action to these two rules, to be executed after the existing actions defined earlier, to forward to forward an initialization packet with $subtype = done$ to the tape switch.

Because we assume FIFO communication between pairs of switches, this packet will arrive at the tape switch *after* all $write$ packets sent from u. On receiving an $done$ message from u, the tape switch is required to contain a rule that matches such a packet returns an initialization packet to u with $subtype = ack$. For every u that is *not* the initialization switch, when u receives an ack packet, it knows the tape switch has processed all of its $write$ messages. Therefore, we require a rule at u that pattern matches such a packet and triggers an action that *modifies* u's local rule that matches $subtype = poll$ and $done = 1$, so that it now forwards the $poll$ packet to the next hop with $done$ still equal to 1.

For the initialization switch itself, on receiving an *ack* packet, it sends a *poll* packet to the next hop in the cycle with $done = 1$. We then require a special extra rule at the initialization switch. This rule matches a *poll* packet arriving from its predecessor in the poll ring with $done = 1$. The action corresponding to this rule is to send the necessary computation packet to the transition switch as required by the definition of implements for our desired configuration. We assume that

this is the only rule matching a *poll* packet arriving with $done = 1$. Therefore, when this rule is triggered the *poll* packet is discarded and there are no initialization messages left in the system.

B. The Initialization Theorem

We now prove that if the network is primed for TM M, and a spark packet arrives at the initialization switch, then after a finite amount of time the system will enter a state in which it implements a configuration of M where M is in its initial state with its read head at the beginning of a tape encoding the switches' forwarding rules for A. At this point, Theorem IV.1 applies to establish the system will subsequently correctly simulate M computation starting from this configuration. Accordingly, this tells us that it is possible to configure an SDN data plane to solve any problems defined with respect to its forwarding rules that can also be solved by a bounded TM.

Theorem V.1. *Assume that at some point t the SDN system is primed for M and the only control packet in the system is a spark packet being sent to the initialization switch. At some future point t', the SDN system will implement the following configuration C_0: M is in its initial state; M's read head is at the beginning of the TM tape; the TM tape encodes all the switches' routing rules for the destinations in A in some fixed encoding format. Furthermore, there are no initialization packets in the system at point t'.*

VI. Related Work

Distributed controllers. To addresses the scalability and fault-tolerance issues that arise when using a centralized controller, there has been a growing body of research exploring the deployment of multiple distributed controllers [7, 11, 15, 16, 22]. Heller et al [11] discuss the controller placement problem and show that introducing k controllers reduces the latency by k for scenarios with frequent network events. However, the improved latency comes with a cost. Levin et al [16] show that having distributed control plane faces consistency challenges and the inconsistencies in the global network view significantly affects the performance (such as optimality) of the network. Almost always, such consistency challenges are either guaranteed using heavy algorithmic machinery (e.g., Paxos-based consensus), or deferred to user applications.

Onix [15] provides an eventually-consistent memory-only DHT (similar to Dynamo [6]) to distribute the network information base (NIB) among the Onix instance. To enforce consistency among these instance, Onix expects the application developers provide inconsistency resolution logic and relies on Zookeeper [12] for coordination. HyperFlow [22] adopts a different approach, in which network states are proactively pushed to all controllers by a publish/subscribe system, thereby enabling individual controllers to locally serve all flows. Hyperflow does not provide consistency guarantees; event reordering and transient inconsistency may occur and are expected to be handled by user applications. In more recent work on distributed control, Dixit et al [7] propose an algorithm for dynamic assignment of switches to controllers, where switches are dynamically handed over from one controller to another as needed. State consistency during the switch handover is guaranteed by a protocol similar to two-phase commitment.

In-network event handling. This work is inspired in part by the SDN communities recent interest in implementing in-network solutions to specific management tasks [21, 2, 3]. (Notice, however, that this existing work leverages OpenFlow's ability to implement conditional rules, whereas our formal model excludes this new functionality.) Outside the SDN realm, Liu et al [17] also promote moving the responsibility for connectivity to the data plane: the authors introduce a data plane mechanism, called Data-Driven Connectivity (DDC), to ensure that a valid forwarding path (to any given destination) is always provided as long as the network is physically connected. Whereas this existing work focuses on specific application scenarios, we study in this paper the fundamental power of this approach.

Computational power. We are not the first to use simulation-techniques to investigate the computational power of restricted network types. Chiesa et al. [5], for example, show how to implement arbitrary logical circuits using networks with routers implementing simple Border Gateway Protocol (BGP) configurations. The implication is that (in theory) BGP can solve the same problems as a Turing Machine. In related work, Emek and Wattenhofer [8], as well as Guerraoui and Ruppert [10], study networks of simple devices (i.e., finite state machines with limited communication capabilities) and show that in both cases these networks can simulate Turing Machines– implying once again a computational power that exceeds their seeming simplicity. Finally, Perešíni and Kostić [20] show how to simulate a Rule 110 cellular automaton in a general network model (though they note that these automaton are only "the first step towards efficiently emulating tape-bounded Turing machines"—the latter being the result we achieve in our SDN model). We adopt this same general approach of simulating a more powerful computational formalism in a restricted network, but the specific techniques we use differ from [20, 5, 8, 10] due to the different assumptions we make concerning our network's operation and/or different goals for the simulation.

VII. Conclusion

The separation of the control plane from the data plane in a SDN brings flexibility and simplicity to network management tasks. Such separation, however, also introduces higher burden on the controllers and longer response time to network events. Recent work has begun to explore the possibility of in-network solutions: that is, delegating the responsibility for handling time-sensitive tasks to the data plane. This paper studies the feasibility of in-network solutions for a broad range of applications. We formally model the capabilities of an SDN data plane, then apply the tools of computability theory to investigate the computational power of a distributed system

with these capabilities. We show, perhaps surprisingly, that such a system can be configured to solve any problem (defined with respect to its current routing rules) that can be solved by a polynomial-space deterministic Turing Machine. The TM simulation in our results, by itself, is not a practical strategy to use in a real network deployment. We instead intend our results to motivate researchers to continue pushing forward in the quest to identify practical applications of in-network solutions in the SDN context.

REFERENCES

[1] Technical report. http://goo.gl/mJhbFb.

[2] M. Borokhovich, L. Schiff, and S. Schmid. Provable data plane connectivity with local fast failover. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2013.

[3] M. Borokhovich and S. Schmid. How (not) to shoot in your foot with sdn local fast failover: A load-connectivity tradeoff. In *Proceedings of the 17th International Conference on Principles of Distributed Systems (OPODIS)*, 2013.

[4] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A nice way to test openflow applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[5] M. Chiesa, L. Cittadini, G. Di Battista, L. Vanbever, S. Vissicchio, et al. Using routers to build logic circuits: How powerful is BGP? In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, 2013.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[7] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed sdn controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2013.

[8] Y. Emek and R. Wattenhofer. Stone age distributed computing. In *Proceedings of the ACM symposium on Principles of Distributed Computing (PODC)*, 2013.

[9] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. *SIGPLAN Not.*, 46(9):279–291, Sept. 2011.

[10] R. Guerraoui and E. Ruppert. Even small birds are unique: Population protocols with identifiers. Technical Report No. LPD-REPORT-2007-006, EPFL, 2007.

[11] B. Heller, R. Sherwood, and N. McKeown. The controller placement problem. In *Proceedings of the First ACM SIGCOMM Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2010.

[13] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, 2012.

[14] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[15] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[16] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized?: State distribution trade-offs in software defined networks. In *Proceedings of the First ACM SIGCOMM Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[17] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker. Ensuring connectivity via data plane mechanisms. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[18] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[19] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *Presented as part of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[20] P. Perešíni and D. Kostić. Is the network capable of computation? In *Proceedings of the International Workshop on Rigorous Protocol Engineering (WRiPE)*, 2013.

[21] L. Schiff, M. Borokhovich, and S. Schmid. Reclaiming the brain: Useful openflow functions in the data plane. In *Proceedings of the Third ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014.

[22] A. Tootoonchian and Y. Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking (INM/WREN)*, 2010.