P5: Policy-driven optimization of P4 pipeline

Anubhavnidhi Abhashkumar* Jeongkeun Lee◦ Jean Tourrilhes† Sujata Banerjee⋆
Wenfei Wu♯† Joon-Myung Kang† Aditya Akella*

*University of Wisconsin-Madison** *Barefoot Networks◦* *HP Labs†* *VMware⋆* *Tsinghua University♯*

Abstract

The physical pipeline of flexible network switches is usually programmed using packet-level programs, such as P4 programs. However, those programs are low level and leave room for further optimization. We propose P5 *(Policy-driven optimization of P4 Pipeline)*, a system that exploits knowledge of application deployments embedded in a high-level policy abstraction to: 1) detect features that are used by applications in a mutually-exclusive way and thereby remove inter-feature dependencies between the tables implementing these features in a network switch. This improves the pipeline concurrency of switches and hence its pipeline efficiency. 2) detect and remove the features that are not used by any application/traffic on the switch in a given topology. This reduces the number of tables and the resource consumed by switches, which also improves its pipeline efficiency. Our experiments on real P4 switch programs show that the resulting switch pipelines are up to 50% more efficient as compared to the cases that do not exploit this information.

CCS Concepts

•**Networks** → **Programmable networks;**

Keywords

Policy Intent; P4; Pipeline Concurrency

1. INTRODUCTION

Building highly flexible programmable networks requires both SDN control plane and data plane programmability. While there have been significant efforts focused on programmable control planes in the past several years, switch data plane programmability is just emerging. Recently, few switch vendors have released flexible, reconfigurable, programmable hardware pipelines: the RMT [11] architecture from Stanford/TI, Intel's Flexpipe [23], and HP 5400Rzl2's ProVision ASIC [6] are some examples of flexible pipeline switches. To program the flexible pipeline, new data plane programming languages are being introduced to enable a user to specify packet processing requirements in logical tables, which are then compiled down to an optimized pipeline of physical hardware stages. One of the first such languages is P4 (Programming Protocol independent Packet Processor) [10]. P4 provides a switch pipeline abstraction on which a programmer can program the entire set of switch features in one P4 program.

In this paper, we make a case for incorporating application-level intelligence into the data plane programming. There are 3 reasons for this.

Standard networking features supported by switches include L2 forwarding, L3 routing, tunneling, multicast, QoS, VLAN, NAT, MPLS, ECMP etc. However, not all traffic will require all features and there are scenarios where disjoint traffic will require different features. For example, operator's management traffic and tenant's data traffic are often processed differently by the network; tenant traffic may get virtualized by VTEP feature but the management traffic wouldn't. Some management traffic may require advanced QoS but telemetry traffic may not. VPN traffic may need stateful firewall but DC internal traffic may not. The knowledge of which application traffic (or tenant) requires which feature is missing from the data plane and may be available as part of high-level policy information.

Secondly, in a network topology with multiple P4 switches, not all features need to be enabled in every switch and some network features can be implemented across the topology. The high-level policies can inform what features need to be enabled in the topology to support all network-wide policies. As long as we can control the routing between switches, we can ensure that traffic that needs a particular feature will traverse the switch that supports that feature [29, 18, 22].

Finally, an important factor affecting the performance capability of today's switches is pipeline concurrency, which allows multiple tables to be placed in the same pipeline stage. This depends on the dependency relationship between tables. A dependency relation exists between two tables (A,B) if table A's match/action impacts the packet processing in ta-

SOSR '17, April 03-04, 2017, Santa Clara, CA, USA
 ISBN 978-1-4503-4947-5/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3050220.3050235

ble B. Two tables can be placed on the same pipeline stage if and only if they do not have a dependency relationship. One of the significant challenges involved in improving the performance of network switches is correctly identifying dependency relationship between tables.

We aim to incorporate application-level intelligence into the data plane, to correctly identify 1) dependencies between tables and 2) features that are safe to remove, thus creating an efficient switch pipeline.

Our paper makes the following contributions:

1. Using high level policies, we create tenant-to-feature mappings, to correctly identify dependencies between the tables, thus creating a more efficient pipeline switch.

2. We further decide which specific features must be supported in the network topology to meet the network-wide policies, and thus enable only those features in selective P4 switches.

3. We demonstrate up to 50% improvement in pipeline efficiency, in terms of number of pipeline stages, with real P4 switch programs.

2. BACKGROUND

In this section, we provide relevant background on high-level policy intent and the P4 language.

2.1 Policy Intent

To provide scalable and effective management for complex networks, several high-level policy intent frameworks have been recently proposed [24, 2, 3, 8, 12, 9]. These frameworks allow defining high level abstract policy intent by hiding physical dependencies and low level details such as IP/Mac addresses, protocols, etc. until the run-time configurations need to be generated. In [24, 2, 3], policy intents can be defined with a group of end points (EPG) and their associated rules. An end point is the smallest unit of abstraction on which a policy is applied, e.g., a server, VM, end-user etc. An EPG is a set of end-points having a common attribute or state, e.g., servers in Campus A. Each EPG has a *logical label* representing various attributes of the member endpoints: e.g., Tenant:Marketing, Location:Campus A, etc. Logical labels, their properties, and relationships can be either *manually* specified by human experts or *automatically* generated by a system like LMS [17]. LMS creates the label namespace and their relationships by analyzing various standard infrastructure data sources such as infrastructure databases, management, controller services, etc. Some examples of these data are OpenDaylight NIC [3], ONF Boulder [12] and OpenStack Congress [4].

Systems like PGA (Policy Graph Abstraction) [24] use labels and label-trees to specify high level policy intents. Figure 1 shows a sample label tree for $Tenant$. In this example, there are three tenants ($Finance$, HR and $Support$), and $Support$ has sub tenants ($Urgent$ and $Normal$). $Urgent$ and $Normal$ are *disjoint* or *mutually exclusive*, which means an endpoint cannot be at both places at the same time. Additionally, $Support$ and $Urgent$ do overlap (parent-child relation), which means all endpoints in $Urgent$ are also in $Support$. In terms of networking, tenant administrators can define which network features are required for which tenant or they can be automatically retrieved from management system or infrastructure database. As shown in Figure 1, the tenant label tree can provide required features as an additional property for each label for tenant-to-feature mappings. In the tree structure, all features are inherited from parents to children nodes. For example, L2 switching and Firewall are required for all tenants. Additionally the Tunnel feature is required for the tenant $Finance$, whereas NAT is required for the tenant HR. Such additional features are exclusive to these tenants, and hence can be classified as *mutually exclusive* features. For example, to handle network traffic for tenant HR, we only need to enable L2 switching, Firewall, and NAT features. In this paper, we use the tenant-to-feature mappings from the label tree to identify mutually exclusive features. By mutually-exclusive features (F_1, F_2), we mean that a packet which requires feature F_1 will not require feature F_2 and vice versa.

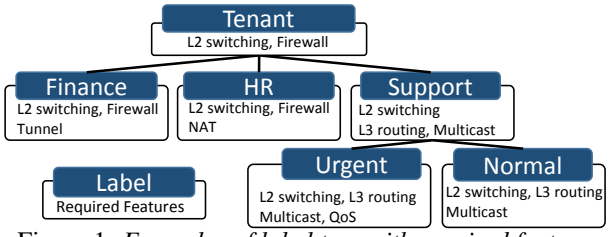

Figure 1: *Examples of label tree with required features*

2.2 P4 Language

P4 [10] is a language to program protocol and target independent packet processing pipelines of switches with programmable data planes. A P4 program consists of headers, parse graph, actions, match-action tables and a control flow between tables. A header defines a sequence and structure of a series of packet fields; a parse graph specifies the header sequences in packets; an action is a customized function to process a packet (composed of primitive actions such as modify, drop, etc.); a table specifies header fields to match and a set of actions to perform - only one of the actions is performed for a matched packet. Finally, a control flow determines the order of tables that are applied to a packet. P4's flexibility and reconfigurability makes it protocol and target independent. That is, many protocols can be described flexibly in headers and parsers, and the program can be compiled to different target systems, ranging from software switches to NPU, FPGA or ASIC based switches.

Figure 2 shows the high level differences between programming conventional fixed pipeline switches and reconfigurable pipeline switches using P4.

RMT [11] and FlexPipe [23] are some examples of high-speed pipeline switch architectures that support the P4 language. In these architectures, logical tables are instantiated onto a pipeline of match-action stages, with each stage having dedicated resources: TCAM, SRAM, ALU, etc. Match-action stages can be separated into ingress and egress pipeline. Efficiently assigning logical P4 tables to a limited number

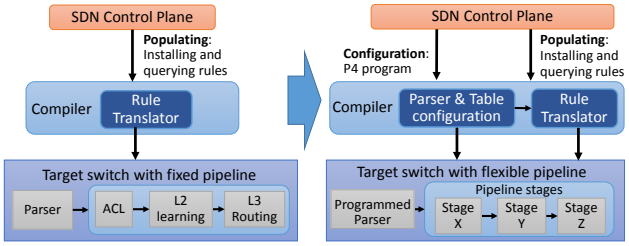

Figure 2: ***Left**: a conventional southbound API that populates pre-determined tables of a fixed pipeline. **Right**: the new trend of the two stage process of 1) defining parser and logical tables in P4 and compiling them into flexible pipeline stages, 2) populating match+action rules in the tables defined by the program.*

of physical stages is a critical job of P4 compilers, and it involves two aspects: table dependency and resource constraints.

A typical P4 program implementing various switch features has more tables than the number of physical stages; Many of the logical tables can be concurrently executed in the same stage as long as they don't have any *dependencies* on one another and the stage has enough resources to run the tables. An efficient compiler packs as many tables as possible in a fewer number of stages, reducing pipeline latency and leaving space to pack more tables and features.

While resource packing has been studied recently in the context of programmable switch pipeline [16, 20], little attention has been given to reducing table dependencies. For example, a standard P4 compiler [5] is composed of 1) a target-independent frontend (`p4-hlir`) that analyzes the given P4 program and expresses table dependencies into a table dependency graph (TDG), and 2) a target-dependent backend that can consume TDG in finding the tables to place in the same resource-constrained stage. Even if the pipeline has large enough net resources across stages, a P4 program may fail to fit in the pipeline if the TDG requires more number of stages than the pipeline provides. Since the pipeline architecture and the number of stages are often static even in a programmable pipeline, detecting and removing unnecessary table dependencies is critical.

Existing P4 compilers [16, 11] detect table dependencies only from a given P4 program, which defines its match-action tables using low-level (packet-level, protocol-level) constructs. In this paper, we propose to identify and remove unnecessary table dependencies by *exploiting knowledge of application requirements and deployments embedded in high-level policy abstractions*.

3. GENERATING ACCURATE TABLE DEPENDENCIES

Currently the P4 compiler alone is responsible for finding dependency relationship between tables. One way to reduce table dependencies, is to leverage policy intents specifying which features will be used by which tenant (*tenant-to-feature mapping*). Systems like PGA [24] use labels, label-mapping and label-trees to specify not only such tenant-to-feature mapping, but also which tenants are mutually exclusive and hence which features are mutually exclusive.

A system like LMS [17] can be used to derive label data from various standard infrastructure metadata sources [3, 12, 4]. Currently there is no way for the P4 compiler to have such information. Using high-level policy information as a source of optimization has not been done in previous language compilers. Our main insight is that we can compute accurate dependency relationships between P4 tables from high-level policy information, by avoiding unnecessary dependencies between tables that belong to mutually-exclusive features. Decreasing the number of dependencies increases the chances of packing more P4-tables to fewer pipeline stages, i.e., it allows more features/tables to fit in the pipeline, thus improving the pipeline efficiency.

Example Scenario: We have a single P4 switch with 3 features enabled: tunnel, NAT and multicast. The P4-tables used by all these features are mentioned in Table 1 and all these tables are reading and/or modifying the destination IP address. According to the label tree and tenant-to-feature mapping given in Figure 1 - tunnel, NAT and multicast are mutually-exclusive features and hence the P4-tables representing these features are also mutually exclusive.

As shown in Table 2, the original P4 compiler would have put all these tables in a separate pipeline stage (five pipeline stages). This happens because according to the P4 compiler, all of the tables have a dependency relationship with every other table as they are all using the destination IP address. On the other hand, the label tree and tenant-to-feature mapping tells us that since all three features are mutually exclusive, there is no inter-feature table dependency and hence the number of pipeline stages can be reduced to three. The two other stages are freed up and can be used to either host new advanced features or increase the size of the standard features placed in stages 1-3.

Feature	P4-tables
Tunnel	*tunnel, ipv4-src-vtep, tunnel-lookup-miss*
NAT	*egress_nat*
Multicast	*outer_ipv4_multicast*

Table 1: *P4 tables used by all features. All these tables are reading and/or modifying destination IP address*

	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
Without mapping	*tunnel*	*ipv4-src-vtep*	*tunnel-lookup-miss*	*egress-nat*	*outer-ipv4-multicast*
With mapping	*egress-nat, outer-ipv4-multicast, tunnel*	*ipv4-src-vtep*	*tunnel-lookup-miss*		

Table 2: *Stage assignment of P4 tables with and without label tree and tenant-to-feature mapping*

4. FEATURE REMOVAL

Fixed pipeline switches often include a large set of features that most customers may not need; typically a feature may be included to support a small set of customers. One of the goals of P4 is to customize the set of features in a switch for a group of customers and avoid installing unused features. However, the programmer that programs the P4 pipeline of the switch may not have access to all policies, and those policies may evolve after switch programming. For example, the programmer may decide to include some features that he thinks might be needed by application policies; however, when the actual policy set is deployed, no policy requires those features. Conversely, at programming time some features from the P4 pipeline may be excluded, to reduce resource consumption, and at deployment time this will prevent the deployment of policies needing them.

Here, knowledge of high level policies is crucial in building an efficient switch data plane with the minimal set of features needed by all the application policies. Thus the set of features to be included in the P4 pipeline can be automatically derived from high level policies. In most P4 programs, most features can be optionally enabled or disabled - for example in the P4 switch program, 13 features can be optionally included with a simple flag. The *tenant-to-feature mapping* of the high-level policy can be used to determine the list of features required for the network. A simple post-processing step can use that list to automatically customize the set of P4 features programmed in all switches.

Removing unused features from the P4 pipeline reduces the resource consumption and in most cases improves performance. Figure 3 represents a complex egress pipeline with 8 network features enabled. The total number of stages in this pipeline is 10. In Figure 3, shaded blocks represent the stages occupied by each feature and the arrows represent the dependency between blocks. One of the main things that stands out is the interdependencies between features. An obvious way to reduce the number of pipeline stages is to reduce the number of features to only the ones required, which is shown in Figure 4. This reduces the number of stages not just because the P4-tables associated with the removed features are missing, but also because the dependencies to those features are avoided and hence they can be placed in an earlier stage. Adding tenant-to-feature mapping can further reduce the number of pipeline stages as shown in Figure 5.

With some policy intents, it might be possible to customize the set of features for each switch in the network topology. However, typical policy intents are decoupled from the underlying topology and do not specify a physical path (i.e. a set of switches), and the physical path may change in complex ways due to fault tolerance, load balancing and QoS. One way to solve this issue is to assume that the (reduced) feature set is the same for all switches. Alternatively, more resource efficiency can be extracted by placing features appropriately in the topology. While we do not explore this in detail, we do present an example scenario in the evaluation and the associated benefits of this approach.

Adapting to policy changes: in case the policy intent changes or the network topology changes, the P4 program may need to be recompiled – with updated feature set and table dependencies – into the switch data plane. To update the data plane under live traffic, as opposed to rebooting the switch, it is also required for the switch control plane software to record various switch states and replay them. Certain software modules may also need to upgrade to work with the new data plane.

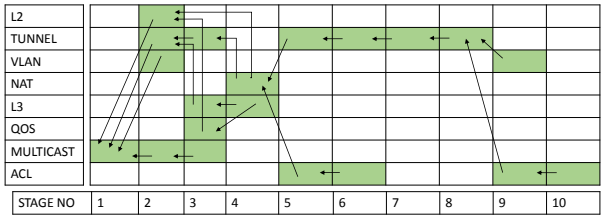

Figure 3: *Egress pipeline with 8 features enabled. An arrow from one block to the other block represents a dependency between the tables in those blocks.*

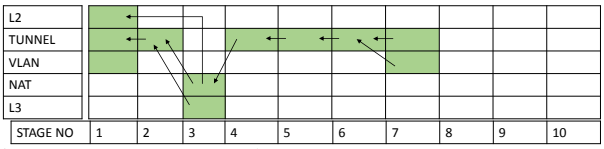

Figure 4: *Egress pipeline after removing four features. The features enabled in this switch are L2, NAT, TUNNEL, VLAN. Number of stages is reduced to 7.*

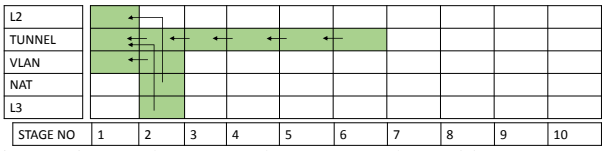

Figure 5: *In this example we are also adding tenant-to-feature mapping: tenant A has (L2,NAT), tenant B has (L2,TUNNEL) and tenant C has (L2,VLAN). Number of stages is further reduced to 6*

Anecdotal information from vendors and operators indicate that the update complexity is higher in the control plane software stack than the data plane. Data plane updates can be as simple as having a new program loaded in memory (or pre-configured in swap resource depending on data plane implementations) and switching from the old to the new after the software setup is complete. Upgrading the software stack takes up to a few seconds [1] while the *data plane update is almost instantaneous, orders of magnitude faster*. Alternatively, network-wide traffic engineering and update scheduling has been a viable solution in production networks [15].

5. EVALUATION

5.1 Experimental Setup

For all our experiments, we use a sample P4 switch program (*switch.p4*) from the P4 github repository [5]. This is the most complex p4 program available. It has 129 tables and over 10K LOC. It describes the data plane of an

L2/L3 switch. Some of the features supported by this sample switch are L2 switching, L3 routing, multicast, tunneling, ACLs, Mirroring, Inband Network Telemetry (INT), MPLS, ECMP, QOS, VLAN, NAT and Unicast RPF. The `p4-hlir` frontend compiler comes with a tool called P4-graph, which generates the table dependency graph (TDG), and the minimum number of ingress and egress pipeline stages required to satisfy all dependencies. From there, P4-graph also assigns a set of tables that can concurrently run together within the same logical stage without considering resource constraints. This logical stage assignment serves as a starting point for backend compilers. We use this tool to identify both the table dependencies and stage assignment of P4-tables.

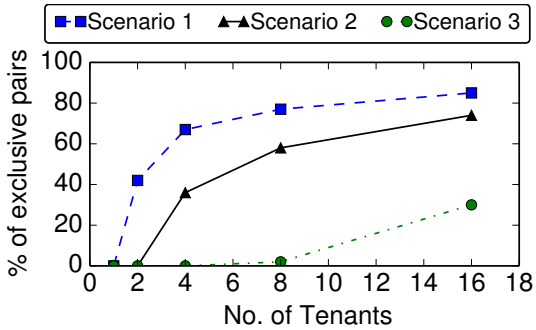

Figure 6: *Percentage of mutually-exclusive pairs vs. number of tenants. The three lines represent three scenarios: each feature used by 1, 2 and 4 tenants, respectively.*

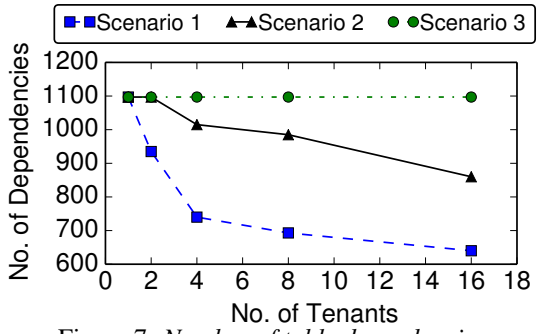

Figure 7: *Number of table dependencies.*

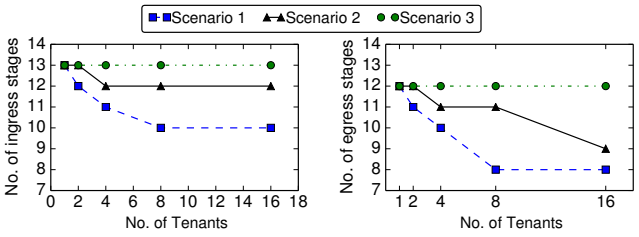

Figure 8: *Number of pipeline stages (both ingress and egress).*

5.2 Dependency reduction

We first show that the tenant-to-feature mapping information can help reduce the number of pipeline stages (both ingress and egress) by reducing the number of table dependencies. In this experiment, we enable 13 different features in the switch, and vary the number of tenants as well as the tenant-to-feature mapping. The L2 feature is an exception and used by every tenant assuming basic L2 switching is the minimally required feature for all traffic. The tenant-to-feature mappings for the other 12 features are controlled as follows.

The ultimate goal of controlling tenant-to-feature mappings is to control the number of mutually-exclusive feature pairs. More precisely, the ratio of mutually-exclusive feature pairs compared to the total number of feature pairs ($C(13, 2)$) is the key metric that affects dependency reductions. To control the metric, we vary the total number of tenants, and also the number of tenants used by each feature (other than L2) in three different scenarios. In *scenario 1*, each feature is used only by one randomly-selected tenant. (Since we are using only 13 features, after increasing the number of tenants to 13, the remaining tenants will have only the L2 feature enabled). In *scenario 2* each feature is used by two randomly-chosen tenants; and in *scenario 3* each feature is used by four tenants (i.e., each tenant uses more features in scenario 3 than in scenarios 1 and 2). Fig. 6 shows the wide range of the metric covered in our experiments: the percentage of mutually-exclusive pairs ranges from zero to 85%. More features become mutually exclusive as the number of tenants increases and each tenant uses fewer features (scenario 1, compared to scenarios 2 and 3).

As expected, making more features mutually-exclusive reduces table dependencies (Figure 7), which then reduces the number of pipeline stages needed to fit the program (Figure 8). For example, the number of egress stages in scenario 1 reduces from 12 down to 8, which is a 33% reduction in terms of stage usage. This also means that up to 50% more tables can fit in the same pipeline assuming similar level of dependencies and resource requirements from the new tables. The 50% increase in pipeline efficiency may mean 1) 50% increase of routing table size or other standard features, 2) room to run additional advanced network functions, or 3) increase of switching throughput, depending on different pipeline architectures.

In all experiments, each data point is an average taken over 20 randomized runs; where in each run, the number of tenants were kept constant and we randomly chose tenants that used each feature.

5.3 Feature removal

In the previous experiment, using tenant-to-feature mapping information did not help reduce table dependencies or stage requirements in *scenario 3*, where tenants have more features in common and thus there are less mutually exclusive feature pairs (see Fig. 6).

We now show that feature removal can reduce the pipeline stages even when features are shared by many tenants. To show the benefit of per-switch feature selection/removal, we consider a simple 2-tier fat-tree DC topology where every ToR switch is connected to every spine. There are the same number of ToR switches and spine switches and we vary the number of ToR-spine pairs. Total number of tenants is fixed to 8, each tenant having two hosts (traffic source and destination). The 16 hosts are randomly assigned their ToR locations in a given topology, thus creating various *tenant-to-*

switch mappings and eventually deriving *feature-to-switch mappings*. A switch needs to implement only features required by the tenant traffic traversing the switch. We analyze only ToR switches in feature selection since ToR switches are typically expected to run much more features than spine switches. As the number of tenants is constant, the set of features required at each switch reduces as the tenants are spread over more number of ToR switches.

In Figure 9, we compute the number of pipeline stages (ingress + egress) needed to pack the required features at each switch and plot the maximum across all switches, i.e. $MAX_{s \in \text{all-switches}}(\ ingress_s + egress_s)$. Each data point is an average taken over 20 randomized runs with different feature-to-tenant and tenant-to-switch mappings. As shown in Figure 9, systematically selecting features to implement at each switch allows removal of unnecessary features (and their tables), thus decreasing the number of pipeline stages even in *scenario 3* where there is no room to optimize table dependencies. We note that the stage reduction benefit gets smaller as the number of switches increases. This is because there are some large features such as tunneling, which alone takes 12 stages. Thus the pipeline length is lower-bounded and there is not much room to further optimize beyond that point.

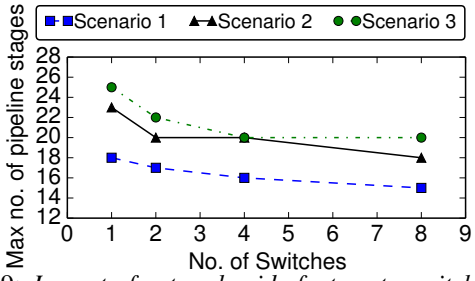

Figure 9: *Impact of network-wide feature-to-switch assignment on max pipeline stages required over all switches. More switches in the network, more room to optimize per-switch feature selection. Total number of tenants is fixed to 8.*

6. RELATED WORK

Many recent approaches have been proposed to provide high-level languages, abstract models, or additional portability layers as a northbound interface for programming entire networks based on policy intent [13, 21, 28, 30, 7, 14, 19, 2, 24]. Most of them have been targeted to control planes (e.g., SDN controllers [13, 21, 28, 30, 20, 7, 14, 19, 2, 24] or cloud controllers (e.g., OpenStack Neutron) [2, 24]), and not to data planes. We have focused on constructing an efficient pipeline switch (as a data plane) using such high-level policies.

P4 [10] has addressed a low-level language for programming packet processors with reconfigurability, protocol independence and target independence. DC.P4 [27] has demonstrated how to use the P4 language for expressing the forwarding plane behaviors of a data center network as a case study. In addition, DC.P4 has contributed improvements to the P4 specification in terms of better language semantics and modularity. Recently, a P4 compiler [16] for reconfigurable switches [16] has defined how to build a switch compiler by using abstractions to hide hardware details and showed results with RMT [11] and Intel's FlexPipe [23]. This P4 compiler used a Table Dependency Graph (TDG) for reinterpreting traditional control and data dependencies in a match+action context. It has also compared the performance of a greedy heuristic design with an Integer Linear Programming (ILP) model.

Ori et al [25] also talk about improving switch performance by packing tables to fewer pipeline stages. But they are only capable of removing dependencies between tables (representing network function boxes) belonging to different service chains.

Concurrent NetCore [26] has specified high-level switch policies as well as concrete, low-level switch architectures. It showed linguistic models of both RMT and Flexpipe architecture. Concurrent NetCore has provided a small number of primitive operations for specifying packet processing, plus combinators for constructing more complex packet processors from simpler ones. There are works [9, 18] that talk about mapping features/state-functions modeled in one-big switch abstraction onto a topology of multiple switches. Nanxi et al [18] considered only table efficiency and not pipeline efficiency. We can use a methodology similar to SNAP's [9] *state placement and routing* to decide feature-to-switch mapping (*feature placement*) - that is, which feature should be enabled in which switch, will be most efficient in terms of total number of pipeline stages traversed across all switches while satisfying all dependency constraints.

7. CONCLUSION

This paper tackles an important challenge that has not received much attention yet on the topic of reconfigurable switch chips. We showed that packet-level programs, such as P4 programs, for flexible switch pipelines are low level, and leave room for further optimization when compiled down to the physical pipeline. Our P5 system exploits knowledge of application requirements and deployments embedded in high level policy specifications to create an efficient programmable switch pipeline. To the best of our knowledge, this is the first system to achieve this goal. The evaluation of our system implementation shows that leveraging the knowledge of high-level policies can provide up to 50% improvement in pipeline efficiency. Our future work will entail a larger study of a variety of P4 programs and scenarios to evaluate the pipeline efficiency that can be achieved using P5. Devising an authoring tool that helps programmers to easily leverage policy/application level information in modular P4 programming is another future work.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work is supported by the Wisconsin Institute on Software-defined Datacenters of Madison and National Science Foundation grants CNS-1302041, CNS-1330308, and CNS-1345249

9. REFERENCES

[1] Miercom Lab Testing Report, Chapter 12 Resiliency during ISSU. http://miercom.com/pdf/reports/20121129.pdf.

[2] OpenDaylight Group Policy. https://wiki.opendaylight.org/view/Group_Policy:Main.

[3] OpenDaylight Network Intent Composition. https://wiki.opendaylight.org/view/Network_Intent_Composition:Main.

[4] OpenStack Congress. https://wiki.openstack.org/wiki/Congress.

[5] P4 github repository. https://github.com/p4lang/.

[6] HP 5400R zl2 Switch Series. *See goo.gl/t3kk9D*, 2015.

[7] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. Netkat: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126, New York, NY, USA, 2014. ACM.

[8] Anu Mercian, Felipe Yrineu, Joon-Myung Kang, Raphael Amorim, Saket M Mahajani, Mario Sanchez and Sujata Banerjee. Network Intent Composition (NIC) Be Feature Update and Demo: Intent Compilation, Lifecycle Management and Automated Mapping. Presented in OpenDaylight Summit 2016, September 2016.

[9] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 29–43, New York, NY, USA, 2016. ACM.

[10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[11] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 99–110. ACM, 2013.

[12] D. L. et al. Open Networking Foundation – Intent NBI - Definition and Principles, 2016.

[13] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, volume 46, pages 279–291. ACM, 2011.

[14] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson. A coalgebraic decision procedure for netkat. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 343–355, New York, NY, USA, 2015. ACM.

[15] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 539–550. ACM, 2014.

[16] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling packet programs to reconfigurable switches. In *USENIX NSDI*, 2015.

[17] J.-M. Kang, J. Lee, V. Nagendra, and S. Banerjee. Lms: Label management service for intent-driven cloud management. In *Integrated Network Management, 2017. IM'17. IFIP/IEEE International Symposium on*. IEEE, 2017.

[18] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the one big switch abstraction in software-defined networks. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 13–24. ACM, 2013.

[19] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 59–72, Berkeley, CA, USA, 2015. USENIX Association.

[20] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 217–230, New York, NY, USA, 2012. ACM.

[21] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software defined networks. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 1–13, Lombard, IL, 2013. USENIX.

[22] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Dream: dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 419–430. ACM, 2014.

[23] R. Ozdag. Intel® ethernet switch fm6000 series-software defined networking. *See goo.gl/AnvOvX*, 2012.

[24] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. Pga: Using graphs to express and automatically reconcile network policies. *SIGCOMM Comput. Commun. Rev.*, 45(4):29–42, Aug. 2015.

[25] O. Rottenstreich, I. Keslassy, Y. Revah, and A. Kadosh. Minimizing delay in network function virtualization with shared pipelines. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):156–169, 2017.

[26] C. Schlesinger, M. Greenberg, and D. Walker. Concurrent netcore: From policies to pipelines. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 11–24. ACM, 2014.

[27] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu. Dc.p4: Programming the forwarding plane of a data-center switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 2:1–2:8, New York, NY, USA, 2015. ACM.

[28] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying sdn programming using algorithmic policies. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 87–98, New York, NY, USA, 2013. ACM.

[29] M. Yu, A. G. Greenberg, D. A. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *NSDI*, 2011.

[30] M. Yu, A. Wundsam, and M. Raju. Nosix: A lightweight portability layer for the sdn os. *SIGCOMM Comput. Commun. Rev.*, 44(2):28–35, Apr. 2014.