pcube: Primitives for network data plane programming

Rinku Shah[*], Aniket Shirke[*], Akash Trehan[*], Mythili Vutukuru, Purushottam Kulkarni

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
{rinku, anikets, atrehan, mythili, puru}@cse.iitb.ac.in

Abstract—**P4 is a domain specific language to configure packet processing pipelines in programmable dataplane switches, and is a powerful idea towards realizing the goal of flexible software-defined networks. This paper presents pcube, a framework that provides a set of primitives to simplify the development of P4-based dataplane applications. pcube provides primitives for loops, summations, and other common operations on indexed state variables, which can be embedded within P4 code and unrolled by the pcube preprocessor. pcube also provides primitives to synchronize state variables across switches in distributed dataplane applications, which are automatically translated into P4 code to send and receive synchronization messages across multiple switches by pcube. We build example dataplane applications such as a distributed load balancer in our framework, and show that using pcube reduces the programming effort (in term of lines of code) significantly—by a factor of up to 5.4x.**

Index Terms—**programmable data plane, P4, programming API**

I. INTRODUCTION

Recent advances in Software Defined Networking (SDN) have expanded the scope of software control on network devices by enabling programmability in the data plane. Research on programmable data planes has come up with new forwarding plane architectures that enable programmability at line rate [1], [2] and languages that can be used to easily program such dataplanes [3], [4]. Researchers have demonstrated that several applications can be significantly accelerated using programmable dataplanes, e.g., key-value caches [5], [6], load balancers [7], [8], network monitoring [9], consensus protocols [10], performance diagnosis [11] and heavy hitter detection [12]. Across all such applications, P4 [3] has emerged as the most popular language for specifying packet processing pipelines on programmable dataplanes. P4 programmers write code to customize packet parsing and the logical flow of packets through match-action tables, all while being agnostic to the hardware platform on which the dataplane will be run. P4 compilers (e.g., [13]) then compile the target-independent P4 code to run on multiple hardware and software programmable dataplane platforms. Several researchers have looked at the problem of optimizing P4-based dataplanes, e.g., by using hardware accelerators to optimize the packet processing pipeline [14], and by caching match-action rules of flows [15]. The key observation of our work is that the problem of easing software development using P4 itself has received lesser attention. ClickP4 [16] proposed a modular approach for P4 programming, using a library of reusable modules that can be easily composed to reduce development time. However, there exists significant scope for automating P4 development effort in dataplane applications.

To understand the problem better, consider a simple load balancing application that assigns incoming flow requests to one of the application backend servers by rewriting the destination IP address of packets. If this application were to be implemented in a programmable dataplane switch, each switch would maintain per-server state (e.g., count of flows currently assigned to the server), and for every incoming request, picks a backend server based on a policy (e.g., the least loaded server). We found that writing P4 code for such an application involves significant repetition of code patterns. For example, P4 does not have an easy way to loop over a set of indexed state variables (e.g., load at the set of backend servers) for initialization or updates. P4 also has no support to perform operations such as summations or finding the minimum/maximum across a set of indexed variables. Further, if the simple load balancer application described above were to be distributed across multiple switches for scalability, the switches would need to exchange information about server load levels amongst themselves in order to balance load suitably. P4 currently has no easy way to synchronize state across switches, and developers must manually write code to generate and process messages to exchange such information. Because P4 can be compiled to a wide variety of platforms, including hardware switches that run at line rate, the language has a minimal set of abstractions that can be supported, and does not support loops and other primitives with unknown execution time. However, the absence of such abstractions to manipulate and synchronize indexed state variables makes P4 code development tedious and prone to manual errors, especially when the number of variables involved is large. Note that the problems described above are not specific to the load balancer application, and similar code patterns are found across a wide range of dataplane applications like heavy-hitter detection, congestion control, and INT (In-band network telemetry).

We propose pcube, *Primitives* for improving *P4 Productivity*, a preprocessor that reduces P4 development effort by providing primitives for easily accessing, manipulating, and synchronizing a set of indexed state variables across

---

[*] Student authors with equal contribution.

distributed programmable dataplane devices. pcube provides abstractions such as loops, min/max computation, summation, and conditional statements over a set of state variables, which are unrolled into regular P4 code by the preprocessor. Further, pcube provides abstractions to synchronize variables across switches, both across a multicast group or with specific switches. Our framework takes the network topology as input, and automatically expands the synchronization primitives into P4 code to send/receive synchronization messages to other switches based on network topology. P4 code generated by the pcube preprocessor is then compiled and deployed on to programmable switches much like regular P4 code.

The main contributions of our work are as follows: (i) the design and implementation of pcube, a preprocessor that provides primitives to ease the development of P4 applications (§II), (ii) evaluation of the benefits of pcube using sample dataplane applications, showing that our framework leads to up to 5.4x reduction in developer effort (§III), and (iii) an opensource codebase of pcube and it's use cases [17].

II. PCUBE DESIGN AND IMPLEMENTATION

A. *The pcube framework*

pcube provides primitives for P4 developers to enable them to write more concise P4 code. Figure 1 shows the overall framework for designing data plane applications with pcube. P4 programmers write dataplane applications using both P4 and pcube constructs. This script, along with network topology information (#switches, #hosts, link information, and switch connection information) is the input to the pcube preprocessor. The preprocessor unrolls the pcube primitives such as loops and converts the user source code to P4 code. In addition to simple unrolling, the preprocessor also creates headers, tables, and actions for pcube synchronization primitives. These headers are used to carry synchronization information between switches, whereas the actions specify the action to be taken upon the receipt of such messages. In addition to P4 code, the preprocessor also generates runtime commands for switch related setup and configuration. If the data plane application runs on multiple switches, we generate separate code for each switch, by considering the topology information as input. The translated P4 code, including logic specified via pcube primitives is then translated to a target specific binary using a standard P4 compiler. The runtime configuration is transferred to the switch via the controller interfaces.

The macro-like framework of pcube is non-intrusive to the basic P4 functionality, and provides various additional abstractions to build custom data plane logic (especially related to event-based inter-switch communication). While an enhancement to P4 itself to support some primitives of pcube is possible, this would require changes to the P4 language specification and the compiler, and is subject to its acceptance across a wide variety of backends that P4 compiles to. The focus of this work is to demonstrate the use of the new primitives and the integration of these features in to the P4 specification is a complimentary effort. Our pcube preprocessor is written in python, and spans around
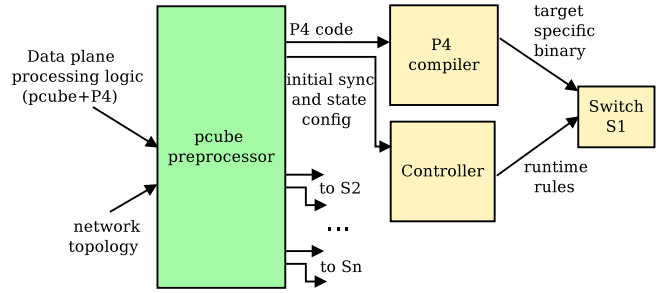

Fig. 1: The pcube framework for data plane programming.

TABLE I: List of pcube primitives.

Type	Annotation	Purpose
Loop	@pcube_for	Iterate over indexed variables
Summation	@pcube_sum	Summation over indexed variables
Minmax	@pcube_minmax	Determine the minimum or maximum value from an input list and choose corresponding action
Conditional	@pcube_cmp	Conditional test over indexed variables
Sync-multicast	@pcube_sync	Share state with multiple switches
Sync-unicast	@pcube_echo	Share state with a specific switch

850 lines of code. pcube is compatible with P4-14, and we are in the process of porting it to P4-16.

Table I shows the list of pcube primitives, which can be classified as: (i) primitives that simplify P4 code within a single switch, (ii) primitives that simplify inter-switch state synchronization. We discuss each of these classes below.

B. *pcube primitives within a single dataplane*

We now describe pcube primitives that can be used to compress repetitive P4 code that accesses a set of similar (indexed) state variables.

Loops. A loop primitive is useful when a set of P4 statements are repeated with different index values. For example, initializing values of switch state, construction of new headers, initialization of arguments, construction of multiple similar tables, and applying the same data plane action on multiple arguments can all be simplified with loops.

Listing 1: The loop primitive.

```
Syntax
@pcube_for (<iterator_var>) (<start>,<end>,<step>)
  // P4 code to be repeated
@pcube_endfor
pcube example
header_type meta_t {
  fields {
      @pcube_for (i) (1,4,1)
          var_counter$i : 32;
      @pcube_endfor
  }
}
pcube preprocessor output
header_type meta_t {
  fields {
      var_counter1 : 32;
      ...
      var_counter3 : 32; //loop unrolled by pcube
  }
}
```

Listing 1 shows an example of using the pcube loop primitive to initialize counters that maintain backend server state in the load balancer.

Summation. The summation primitive is useful when the sum of a list of state variables is required. Listing 2 shows code where the switch drops a packet when the cumulative load on the servers exceeds threshold.

Listing 2: The summation primitive.

```
Syntax
@pcube_sum (<start>,<end>,<step>) (<iterator_var>)
pcube example
if ((@pcube_sum(1,4)(var_counter$i)) > 10000){
  apply(drop);
}
pcube preprocessor output
if ((var_counter1 + var_counter2 + var_counter3) > 10000){
  apply(drop);
}
```

Minmax. This primitive is useful in cases when the minimum or maximum from a set of switch state arguments has to be determined, and a corresponding action has to be applied if the condition is satisfied. The programmer specifies the desired operator ($< | <= | > | >=$), and specifies the action for each argument that can potentially satisfy the min-max condition. For example, Listing 3 shows load balancer code where the switch determines the least loaded server (i.e., the one with the minimum var_counter$i) upon arrival of a new flow, and forwards the packet to that server.

Listing 3: The minimax primitive.

```
Syntax
@pcube_minmax (<relop>)
  @pcube_case (<var$1>):
      //code to be executed if var$1 is min or max
    @pcube_endcase
  ... //code for 'i' other cases
@pcube_endminmax
pcube example
@pcube_minmax (<=)
  @pcube_case var_counter1:
      apply(tab_server1);
  @pcube_endcase
  ...
  @pcube_case var_counter3:
      apply(tab_server3);
  @pcube_endcase
@pcube_endminmax
pcube preprocessor output
if(var_counter1 <= var_counter2 and
   var_counter1 <= var_counter3){
    apply(tab_server1);
}
else
if(var_counter2 <= var_counter1 and
   var_counter2 <= var_counter3){
    apply(tab_server2);
}
else
if(var_counter3 <= var_counter1 and
   var_counter3 <= var_counter2){
    apply(tab_server3);
}
```

Multi-condition. This primitive is useful to test the same condition across a set of switch state variables. Listing 4 shows an example of the load balancer application, where the switch drops an incoming packet if it finds that all the servers are overloaded (beyond 1000 connections in this example). We compare a set of variables provided as argument with a constant variable using the specified relational operator, the condition is applied for all variables in the set, and all conditions are concatenated using the and/or logical operator.

Listing 4: The multi-condition primitive.

```
Syntax
@pcube_cmp(<start>,<end>,<step>)(<relop>)(<logop>)(<
    loop_argument>,<constant>){
  //code
}
pcube example
if (@pcube_cmp (1,4,1)(>=)(and)(var_counter$i,1000)){
  apply(drop);
}
pcube preprocessor output
if (var_counter1 >= 1000 and var_counter2 >= 1000 and
    var_counter3 >= 1000){
  apply(drop);
}
```

C. pcube primitives for distributed dataplanes

Multiple switches in a distributed dataplane application frequently need to communicate with each other, in order to exchange state variables, or reply to requests. We envisage two modes of communication between switches: a *multicast* synchronization mode, where a switch broadcasts a *sync message* to a subset of switches in order to request or provide some state information, and a *unicast* mode where a switch communicates with one other switch.

Multicast synchronization. To automatically generate P4 code to synchronize a set of state variables using pcube, the developer specifies the condition to trigger the synchronization, the state variables to be synchronized/communicated to other switches, and a custom/unused packet header field that can be used to identify these special synchronization messages.

Listing 5: Multicast synchronization primitive.

```
Syntax
if (<condition>){
  @pcube_sync(<custom_header_field>,<field_value>)
    <sync_var$1>
    ...
    <sync_var$n>
  @pcube_endsync
}
pcube example
if (var_counter < THRESHOLD){
  @pcube_sync(pkt.type,2) //pkt.type=2: sync_update packet
    var_counter
  @pcube_endsync
}
pcube preprocessor output
if (var_counter < THRESHOLD){
  apply(sync_tab); //@pcube_sync replaced
}
```

For example, Listing 5 depicts example code to synchronize the server load variables across a set of switches in our simple load balancer example, with the synchronization being triggered only if a switch perceives that its utilization falls below a threshold. When the pcube preprocessor encounters this primitive, it generates P4 code to lookup a special sync action table that is generated by pcube (and must be included in the P4 code by the developer). The default action in this sync action table creates a special sync message by cloning the received packet, and pushes multiple header fields that embed

the state variables that must be communicated to the other switches. This action code also sets the header field to identify this packet as a special sync message, using the field indicated for this purpose by the programmer. Whenever the condition to trigger synchronization occurs at runtime, the code generated by pcube takes care of automatically generating synchronization messages to multiple switches. pcube takes the network topology identifying the multicast group of switches as input, and generates runtime configuration for creation of multicast identifiers, handles and their associations, which ensures that the sync message is delivered to specific switches in the multicast group.

Unicast synchronization. This primitive is similar to the multicast synchronization primitive, except that the state values are shared as response only to the source node that requested the synchronization (hence the name "echo"). Listing 6 shows an example from our simple load balancer, where a switch sends its state variables as response when it receives a request message from another switch. The implementation of this primitive in pcube is similar to that of the multicast primitive, with the difference that a response packet is sent back only to the source by adding a runtime command mirroring the packets back to the source.

Listing 6: Unicast synchronization primitive

```
Syntax
if (<condition>){
  @pcube_echo(<custom_header_field>,<field_value>)
    <sync_var$1>
    ...
    <sync_var$n>
  @pcube_echosync
}
pcube example
if (pkt.type == 1){ //pkt.type=1: echo_request packet
  @pcube_echo(pkt.type,2) //pkt.type=2: echo_update packet
    var_counter
  @pcube_endecho
  }
pcube preprocessor output
if (pkt.type == 1){
  apply(echo_tab); //@pcube_echo replaced
}
```

D. Nesting of pcube primitives

pcube allows nesting of the pcube primitives. The pcube preprocessor is designed as a multi-pass program and performs four passes on an input pcube program. The pcube primitives are converted to corresponding P4 code according to the following order, where primitives later in the order can nest the primitives preceding it: loops, summation, multi-condition/minmax, and sync/echo synchronization primitives. The parsing order is decided to enable useful nesting of pcube primitives. Listing 7 shows the nested pcube code for the example code in Listing 3.

Listing 7: Nested Pcube example

```
@pcube_minmax (<=)
  @pcube_for (i) (1,4,1)
      @pcube_case var_counter$i:
          apply(tab_server$i);
      @pcube_endcase
  @pcube_endfor
@pcube_endminmax
```
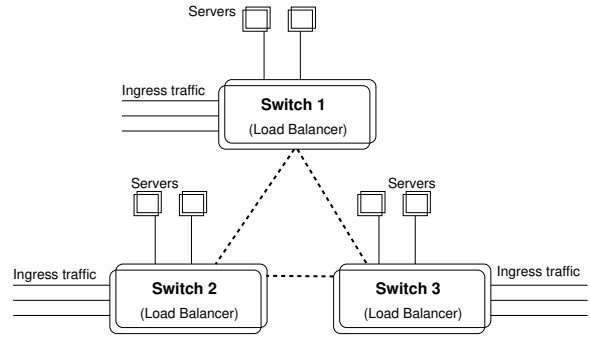

Fig. 2: Load balancer application: example topology.

III. USE CASES

We now evaluate pcube by implementing two sample applications in our framework: a simple distributed load balancer, and a heavy hitter detection application. We evaluate two aspects of our framework: the number of lines of code that can be saved using pcube primitives, and the correctness of our primitives.

A. Load Balancer

Our load balancer application runs at multiple dataplane switches. It assigns new incoming flows to one of the multiple backend servers, and the switches hosting the load balancer application rewrite the destination IP addresses of packets to redirect them to the selected backend. For scalability, the pool of backend servers is distributed amongst the switches. A switch assigns new flows to one of the servers in its "local" pool as long as these servers are not overloaded. Our load balancer keeps track of the current number of flows processed as the load for each server. When a new flow request arrives, the server with minimum flow count is allocated, and server flow counter is updated. When all local servers of a switch are overloaded, it proceeds to assign flows to one of the remote servers. We maintain a table that stores the flow hash and the server allocated for the flow, so that all the packets of the same flow are forwarded to the same server. If the switch receives the last flow packet, the route table entry for the corresponding flow is deleted, and the server flow counter is updated. Figure 2 illustrates our simple load balancer, where three switches manage of a pool of two local servers each, and hosts generate traffic to the servers.

In order to learn about the load levels of remote servers for efficient load balancing, the switches in our application synchronize the load levels of their local pool of servers with other switches in two cases: (i) if a switch finds that all its local servers are overloaded, it generates a sync message requesting the load levels from other switches, so that it can find a lightly loaded remote server for future flows, and (ii) if a switch finds that it is very lightly loaded, it sends a sync message advertising its low load levels to other switches, so that they may redirect flows to its local pool. The threshold load levels at which these synchronization messages are sent are configured by the P4 programmer at runtime. Note that

we avoid continuous synchronization of load levels of all servers across all switches in order to limit the amount of synchronization traffic.

We implemented this load balancer application using pcube. We now show snippets of code from our implementation to illustrate the usefulness of our primitives. Listing 8 shows a snippet of code where a switch tries to find a local server that is not overloaded to assign as a backend for a new flow, failing which it triggers a probe synchronization message to all servers requesting them to send the load levels of their local servers as reply. This snippet uses the multi-conditional, summation, and multicast synchronization primitives. Next, Listing 9 shows a code snippet where a switch that has received the probe described above, and replies with its own load level, illustrating the unicast synchronization primitive. Once a switch has learnt of the load at remote servers via replies to its probes, Listing 10 shows how a switch drops new flows if it finds all servers (including remote ones) to be overloaded, or directs flows to the least loaded remote server if any of them is found to be serving a load below a threshold. This snippet illustrates the loop, min/max, and multi-conditional primitives. Finally, Listing 11 shows how a switch with an underloaded pool of servers announces its load levels to other switches to enable them to direct traffic to it.

Listing 8: Local switch forwarding.

```
if(@pcube_cmp(0,NUM_SERVERS,1)(<)(or)(flow_count$i,THR1)){
  apply(route_to_local_server_table);
  if((@pcube_sum(0,NUM_SERVERS,1)(flow_count$i))>THR2)){
    apply(probe_table);
 }
} else ...
if(probe == 1){ // send sync request
  @pcube_sync(pkt.type,1)
  @pcube_endsync
}
```

Listing 9: Process sync request at switch.

```
if (pkt.type == 1){
    @pcube_echo(pkt.type ,2)
        @pcube_sum (0,NUM_SERVERS,1)(flow_count$i)
    @pcube_endecho
}
```

Listing 10: Remote switch forwarding.

```
if (@pcube_cmp(0,NUM_SWITCHES,1)(>=)(and)(flow_count$i,THR3
    )){
 apply(drop_table);
}else {
    @pcube_minmax (<=) // find least loaded switch
    @pcube_for (i) (0,NUM_SWITCHES,1)
      @pcube_case switch_flow_count$i:
        apply(switch$i_route_table);
      @pcube_endcase
    @pcube_endfor
    @pcube_endminmax
}
```

Listing 11: Send sync update from switch.

```
// Sync update message
if ((@pcube_sum(0,NUM_SERVERS,1)(flow_count$i))<THR2){
    @pcube_sync(pkt.type,2)
      @pcube_sum(0,NUM_SERVERS,1)(flow_count$i)
    @pcube_endsync
  }
}
```
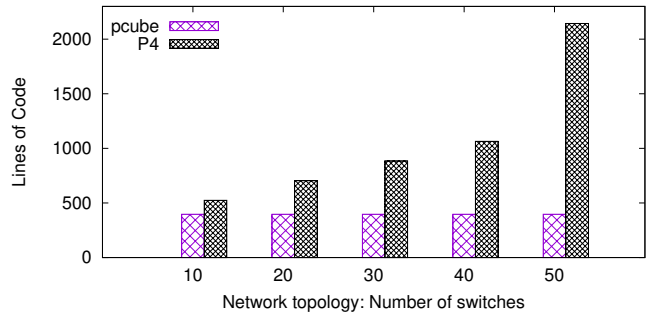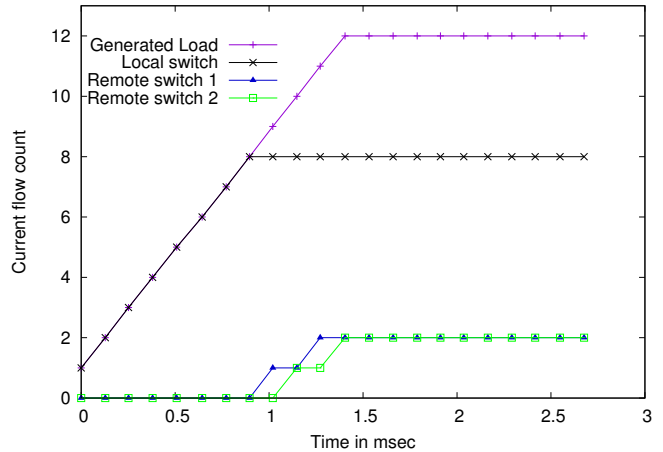

Fig. 3: pcube vs P4: Distributed load balancer.

Fig. 4: Load balancer: 3 switches, switch threshold = 8 flows.

Savings in developer effort. We implement the load balancer application described above in pcube, and compiled the resulting P4 code to the bmv2 [18] target. We then emulated a network with varying number of nodes using mininet [19], and generated flows to our load balancer using scapy [20]. We varied the network topology by varying the number of nodes (servers, hosts, and switches) in the network, and generated P4 code using pcube for different network topologies. Figure 3 shows the number of lines in the load balancer application when using pcube, and when directly writing P4 code. We find that while the size of the P4 code base increases with increasing nodes in the network topology, the size of the pcube code remains constant due to the pcube primitives that provide an efficient way to access and update indexed state variables. We see from the graph that we achieve up to 5.4X reduction in LoC, indicating significant reduction in developer effort with pcube.

Correctness. We now show that our load balancer is able to correctly balance load to multiple servers. Specifically, we evaluate our synchronization primitives and whether the messages are responses are generated as expected by the autogenerated pcube code. We consider a topology with 3 switches, 2 local servers per switch and a single host generating traffic. The lower limit threshold configured is 3 flows,
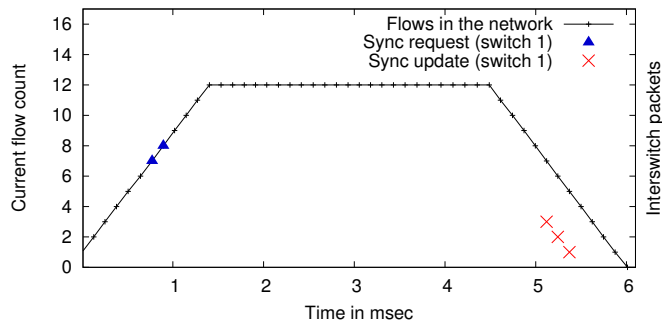
Fig. 5: Interswitch messages generated by the ingress switch.

and the upper limit threshold is configured as 6. The client generates load at the rate of one new flow every 0.128 seconds to one of the three switches. We configure the switch to move from its local pool of servers to start probing other switches when the cumulative load on its servers crosses a threshold of 8 flows. Figure 4 shows the timeline of the flow arrivals and assignments, and Figure 5 shows the synchronization messages generated by the ingress switch, to which the host sends traffic. We see from the Figure 5 that once the server load crosses the configured threshold, probe messages are automatically generated by pcube, resulting in the switch learning of the load at the remote servers and diverting traffic to them.

B. Heavy hitter detection

The heavy hitter detection application [21] identifies sources of traffic that generate huge amounts of traffic and drops packets from such sources. Implementing this application in the dataplane of a programmable switch is complicated by the fact that there is not enough storage to maintain state for all sources. Therefore, optimizations such as a counting bloom filter are used to efficiently store counts for heavy hitter sources without requiring $O(n)$ space. The code for the counting bloom filter involves computing and updating multiple hashes and updating the corresponding indices of a data structure. Such code has significant potential to be compressed with pcube primitives. We started with a heavy hitter detection application [21] and rewrote it using pcube primitives. We found that we could reduce the number of lines of code from 676 to 191, a reduction of 3.53x, when the bloom filter bucket size was 50, and a reduction of 1.44x, when the bloom filter bucket size was 10. This exercise shows that pcube is very useful in reducing the effort of developing realistic programmable dataplane applications.

IV. Conclusion

This paper presented the design and implementation of pcube, a preprocessor framework that simplifies the development of P4 code by providing primitives for loops, summation, multi-conditionals, min/max comparisons and abstractions to synchronize state across multiple switches. Based on the network topology, pcube primitives are translated to P4 code

and further compiled and deployed on multiple dataplane switches of a distributed application. We demonstrated the usefulness of pcube by implementing a set of sample distributed data plane applications, and reduce programming effort (in term of lines of code) significantly—by a factor of 5.4x for the load balancing application and reduction factor directly proportional to the granularity of flow categories for heavy hitter detection application.

As part of future work, we plan to expand the set of primitives to include handling of dynamic changes to network topology and network failures.

References

[1] 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 *Proc. of the ACM SIGCOMM Conference*, 2013.
[2] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, "drmt: Disaggregated programmable switching," in *Proc. of the Conference of the ACM SIGCOMM Conference*, 2017.
[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Computer Communication Review*, July 2014.
[4] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *Proc. of the ACM SIGCOMM Conference*, 2016.
[5] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proc. of SOSP*, 2017.
[6] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "Kv-direct: High-performance in-memory key-value store with programmable nic," in *Proc. of SOSP*, 2017.
[7] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proc. of the SoSR*, 2016.
[8] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proc. of the ACM SIGCOMM Conference*, 2017.
[9] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *Proc. of the ACM SIGCOMM Conference*, 2017.
[10] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "Netpaxos: Consensus at network speed," in *Proc. of the ACM SIGCOMM SoSR*, 2015.
[11] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data plane performance diagnosis of tcp," in *Proc. of the SoSR*, 2017.
[12] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proc. of the SoSR*, 2017.
[13] "P4c github page," https://github.com/p4lang/p4c, 2017.
[14] A. Abhashkumar, J. Lee, J. Tourrilhes, S. Banerjee, W. Wu, J.-M. Kang, and A. Akella, "P5: Policy-driven optimization of p4 pipeline," in *Proc. of the SoSR*, 2017.
[15] Z. Ma, J. Bi, C. Zhang, Y. Zhou, and A. B. Dogar, "Cachep4: A behavior-level caching mechanism for p4," in *Proceedings of the SIGCOMM Posters and Demos*, ser. SIGCOMM Posters and Demos, 2017.
[16] Y. Zhou and J. Bi, "Clickp4: Towards modular programming of p4," in *Proc. of the SIGCOMM Posters and Demos*, 2017.
[17] "pcube project," https://github.com/networkedsystemsIITB/pcube, 2018.
[18] "Behavioral-model," https://github.com/p4lang/behavioral-model, 2017.
[19] "Mininet," http://mininet.org/, 2017.
[20] "Scapy github page," https://github.com/secdev/scapy, 2017.
[21] "SIGCOMM P4 tutorials," https://github.com/p4lang/tutorials, 2017.