Elastic Switch Programming with P4All

Mary Hogan
Princeton University

Shir Landau-Feibish
Princeton University

Mina Tahmasbi Arashloo
Cornell University

Jennifer Rexford
Princeton University

David Walker
Princeton University

Rob Harrison
U.S. Military Academy, West Point

ABSTRACT

The P4 language enables a range of new network applications. However, it is still far from easy to implement and optimize P4 programs for PISA hardware. Programmers must engage in a tedious "trial and error" process wherein they write their program (guessing it will fit within the hardware) and then check by compiling it. If it fails, they repeat the process. In this paper, we argue that programmers should define *elastic* data structures that stretch automatically to make use of available switch resources. We present P4All, an extension of P4 that supports elastic switch programming. Elastic data structures also make P4All modules reusable across different applications and hardware targets, where resource needs and constraints may vary. Our design is oriented around use of symbolic primitives (integers that may take on a range of possible values at compile time), arrays, and loops. We show how to use these primitive mechanisms to build a range of reusable libraries such as hash tables, Bloom filters, sketches, and key-value stores. We also explain the important role that elasticity plays in modular programming, and we allow programmers to declare utility functions that control the relative share of data-plane resources apportioned to each module.

ACM Reference Format:
Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. 2020. Elastic Switch Programming with P4All. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20), November 4–6, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3422604.3425933

1 INTRODUCTION

For the past several decades, innovation in computer networking has been painfully slow. Thanks to the closed, fixed-function devices deployed in operational networks, the path from idea to implementation and deployment has been long and arduous. Now this is changing. The advent of high-speed programmable data planes, and programming languages like P4 [2, 24], enables new ideas to come to fruition quickly.

However, despite making it *possible* to program the network, P4 does *not* make it *easy*. To process packets at high speed, P4-capable devices impose restrictions on processing and memory resources, and these restrictions are specific to each target device. Programmers must grapple with these low-level resource limitations directly in writing their P4 programs, never sure if their programs can even "fit" on a given target, let alone use the available resources effectively. They typically must iterate repeatedly, adding or removing blocks of code manually, or adjusting magic constants.

The problem compounds when the programmer writes sophisticated applications that combine multiple kinds of functionality. Many applications suitable for production networks must combine multiple functions, such as, traffic monitoring, packet forwarding, and access control. Each of these requires different kinds of computation and data structures, yet deciding how much precious switch resources to allocate to each function remains tedious and error-prone. Worse yet, programmers cannot easily reuse common modules across different applications and targets, because the resource needs and constraints change from one context to the next. Although Figure 1 shows that many applications use common data structures and could benefit from code reuse, these structures are rarely shared between developers as libraries due to the lack of modularity. Instead, they must be rewritten to fit the constraints of a particular application. Of course, with more code rewriting, comes more developer time, and worse, more bugs. In the long run, such a writing and rewriting methodology is likely to lead to less reliable networks.

P4 was intended to be a reusable language, with a single program able to compile to many targets. While this is not the case today, we believe that it is both possible and necessary. P4 applications should only have to be written once, not rewritten for each programmable target. In other words, the programming language should not be tied to a target's resources. The learning curve for programming in P4 is significantly steeper because programmers must be so cognizant of a target's constraints. Rather than the programmer incorporating target-specific constraints into an application's code, the target's compiler should allocate available resources to each piece of the application. The isolation of a target's properties from the language crucially eases the burden of the programmer.

Our solution is to extend the P4 language with the ability to write *elastic* programs. An elastic program is a single, compact program that can "stretch" to make use of available hardware resources. Elastic programs can be constructed from any number of elastic modules that each stretch arbitrarily to fill available space. For example, consider the NetCache application [15] that caches popular keys by combining (i) a count-min sketch [5] (to track key popularity) and (ii) a key-value store (to store and serve popular keys), both of which consume finite switch memory. An elastic NetCache application may be constructed from an elastic count-min sketch and an elastic key-value store. To control the relative stretch of these modules, the programmer can specify a utility function that the

HotNets '20, November 4–6, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8145-1/20/11…$15.00
https://doi.org/10.1145/3422604.3425933

Module	Used in
Key-value store/ hash table	Precision [1], Sonata [8], Network-Wide HH [9], Sketchvisor [11], LinearRoad [13], NetChain [14], NetCache [15], FlowRadar [18], HashPipe [28], Elastic Sketch [31]
Hash-based matrix	SketchVisor [11], NetCache [15], Nitrosketch [19], UnivMon [20], Sharma et al. [25], Fair Queueing [26], Elastic Sketch [31]
Hierarchical sketch	SketchLearn [12]
Bloom filter	NetCache [15], FlowRadar [18], SilkRoad [21], Sharma et al. [25]
ID indexed table	Blink [10]

Figure 1: PISA data structures

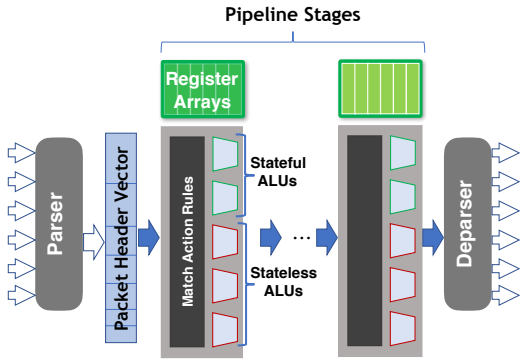

Figure 2: Protocol Independent Switch Architecture (PISA)

compiler maximizes. The NetCache application could maximize the cache "hit rate" by allocating additional memory for the key-value store (to store more of the "hot" keys) while ensuring that enough remains for the count-min sketch to produce sufficiently accurate estimates of key popularity. In addition to memory, programs could simultaneously maximize the use of other switch resources such as available arithmetic logic units and pipeline stages.

To implement these elastic programs, we present P4All, a backward-compatible extension of the P4 language with four key additional features: (1) symbolic values, (2) symbolic arrays, (3) bounded loops with iteration counts governed by symbolic values, and (4) integer linear utility functions. Symbolic values make the sizes of arrays and other data structures flexible, allowing them to stretch when necessary. Loops indexed by symbolic values make it possible to operate on variably-sized, elastic data structures. Utility functions allow the programmer to inform the P4All compiler how to optimize the allocation of limited data-plane resources, and how to prioritize the resource needs of one elastic module over another.

To support these new features, the P4All compiler would need to map an elastic data structure onto a fixed-size programmable switch. In other words, the compiler would determine the symbolic values and array lengths while optimizing the provided utility function. Today's P4 compilers already use optimization to compile to PISA switches (e.g., [16]). We believe that by adding steps to optimize for elastic parameters, we can adapt these existing compilers to P4All.

In this paper, we begin by discussing the difficulty of programming for PISA (§ 2). We then describe the constructs P4All provides to remedy these problems (§ 3). Next, we briefly sketch the P4All compiler (§ 4). The paper ends with a discussion of related work (§ 5), and future research directions (§ 6).

Ethics: This work does not raise any ethical issues.

2 PISA PROGRAMMING PERILS

Writing P4 programs for PISA switches is hard. In this section, we review the many switch resource constraints that programmers must consider, and then illustrate how they make P4 programming so difficult.

2.1 PISA Resource Constraints

P4 is designed to program a *Protocol Independent Switch Architecture* (PISA) data plane (Figure 2). This architecture contains a programmable packet parser, a processing pipeline, and a deparser. Between parser and deparser sits the packet-processing pipeline, which contains fixed resources that bound the computation performed on a packet.

- **Pipeline stages.** The processing pipeline is composed of a fixed number of stages.
- **Registers.** A stage is associated with a limited number of registers (of limited width) that serve as persistent memory.
- **Match-action rules.** Each stage stores match-action rules in either TCAM or SRAM. When a packet's fields match a rule, an associated action is performed.
- **ALUs.** Actions are performed by the ALUs associated with a stage. The numbers of stateful ALUs (that perform actions requiring registers) and stateless ALUs are limited.
- **Packet header vector (PHV).** The PHV that carries information from packet fields and additional per-packet metadata through the pipeline has limited width.
- **Recirculation bandwidth.** A program may recirculate a packet by sending it back to the beginning of the pipeline once it leaves the last stage. If too many packets are recirculated, throughput is decreased.

2.2 Programming Under Resource Constraints

Programming PISA devices is difficult because the resources available are so limited. The architecture forces programmers to keep track of implicit dependencies between actions, to try to lay out those actions across stages, to compute memory requirements of each task, and to fit the jigsaw pieces emerging from many independent tasks together into the overall resource-constrained puzzle of the pipeline. Although the compiler does these tasks, the programmer is forced to essentially do the work of the compiler by themselves to construct a P4 program that "fits" well in a given target.

The P4 language attempts to alleviate a number of these problems by providing a layer of abstraction above PISA. However, experience with P4 programming suggests, that while a good start, the language is simply *not abstract enough*. It asks programmers to make fixed choices ahead of time about the size of data structures and the amount of computation the programmer believes the compiler can squeeze onto a particular switch. These are difficult jobs to do well, even for experts, and next to impossible for novices. Inevitably,

attempts at estimating resource bounds leads to some amount of trial and error. Unfortunately, current P4 compilers are not terribly fast and when compilation fails, feedback is limited. Hence, the program-debug-optimize cycle is slow. In summary, the current development environment requires a lot of fiddly, low-level work and takes human time and energy away from innovating at a high level of abstraction.

To illustrate some of the difficulties of programming with P4, consider an engineer in charge of upgrading their network to use a new caching subsystem, based on NetCache [15], which is designed to serve cached items in the switch. NetCache contains two main data structures, a *count-min sketch* (CMS) for tracking the popularity of keys, and a *key-value store* to map popular keys to values.

First, the engineer focuses on implementing the CMS, a probabilistic data structure that uses multiple hash functions to keep approximate frequencies for a stream of items in sub-linear space. Intuitively, CMS is a two-dimensional array of w columns and r rows. For each packet (x) that enters the switch, its flow ID (f_x) is hashed using r different hash functions ($\{h_i\}$). To approximate the number of times flow f_x has been seen, one computes the minimum of the values stored in columns $h_i(f_x)$ for all r rows.

Figure 3 presents a fragment of the P4 program developed by our engineer to create a CMS. Lines 1-7 declare the metadata needed to look up estimates in the CMS for a particular packet with a specific flow ID. Lines 10-13 declare the low-level data structures (registers) that actually make up the CMS–four rows ($r = 4$) of columns ($w = 2048$) that each store values of 32 bits. Lines 15-21 and 23-27 define the actions for hashing and updating values in the CMS. The hashing action is a complex action containing several atomic actions: (1) an action to hash the key to an index in a register array, (2) an action to increment the count found at the index, and (3) an action to write the result to metadata for use later in finding the global minimum. As our engineer adds more of these actions, it becomes increasingly difficult to estimate resource requirements. In the *apply* fragment of the P4 program (lines 29-45), which details the actions and/or rule tables applied to a packet, the program first executes the hash actions, computing and storing counts for each hash function, and then compares counts, looking for the minimal one. To determine if this allocation actually works on the target, the engineer must write and compile the full program. If it fits, then great; if not, the engineer enters a tedious "trial and error" process of rewriting and recompiling the program.

Upon reviewing the code for the CMS in the NetCache application, some of the deficiencies of P4 are immediately apparent. First, there is a great deal of repeated code: repeated data structure definitions, repeated action definitions, and repeated invocations of those actions in the apply segment of the program. Good programming languages make it possible to avoid repeated code by allowing programmers to craft reusable abstractions that can encapsulate the behavior of many similar statements. Avoiding repetition in programming has all sorts of good properties including the fact that when errors occur or changes need to be made, they only need to be fixed/made in one place. This not only saves time but helps avoid subsequent errors. Effective abstractions also help programmers change the number or nature of the repetitions easily. Unfortunately, P4 is missing such abstractions. One might also notice that

```
1  struct custom_metadata_t {
2      bit<32> min;
3      bit<32> index0;
4      bit<32> count0;
5      ...
6      bit<32> index3;
7      bit<32> count3; }
8  control Ingress( ... ) {
9      /* register array for each hash table */
10     register<bit<32>>(2048) counter0;
11     . . .
12     register<bit<32>>(2048) counter3;
13    /* an action to update each hash table */
14     action incr_0() { . . . }
15     . . .
16     action incr_3() { . . . }
17     /* an action to set the minimum */
18     action min_0(){meta.min = meta.count0;}
19     . . .
20     action min_3(){ . . . }
21     /* apply to each packet */
22     apply {
23         /* initialize global min */
24         meta.min = 0;
25         /* compute hashes */
26         incr_0();
27         . . .
28         incr_3();
29         /* compute minimum */
30         if (meta.count0 < meta.min) {min_0();}
31         . . .
32         if (meta.count3) < meta.min) {min_3();}
33         } }
```

Figure 3: Count-Min Sketch in P4 (16)

the programmer had to choose magic constants (like 2048) and test themselves if such constants lead to programs that can compile.

3 ELASTIC PROGRAMMING IN P4ALL

To prevent the problem of repeated code and the tedious "trial and error" process, we argue for the use of *elastic data structures*. These data structures may be developed modularly in separate libraries and then combined, off-the-shelf, to help users quickly and easily build efficient new applications, like an elastic NetCache. We present P4All as an extension to P4 that enables elastic programs.

To build elastic applications, programmers use the following four-step design methodology: (i) declare the elastic parameters, (ii) construct elastic data structures, (iii) define elastic operations, and (iv) manage competing resource needs. We illustrate this process by examining the definition of an elastic count-min sketch and its use in an elastic NetCache application.

3.1 Declare the Elastic Parameters

The first step in defining an elastic data structure is declaring the parameters that control the "stretch" of the structure. In the case of the CMS there are two such parameters: (1) the number of rows in the sketch (*i.e.*, the number of hash functions), and (2) the numbers of the columns (*i.e.*, the range of the hash). Such parameters are defined as *symbolic values*:

```
symbolic int rows;
```

```
/* Count-min sketch module */
symbolic int rows;
symbolic int cols;
assume 0 <= rows && rows < 4;
struct custom_metadata_t {
    bit<32> min;
    bit<32>[rows] index;
    bit<32>[rows] count; }
register<bit<32>>(cols)[rows] cms;
action incr()[int index] { ... }
action min()[int index] { ... }
control hash_inc( ... ) {
    apply {
        for (i < rows) {
         incr()[i]; }
    } }
control find_min( ... ) {
    apply {
        for (i < rows) {
            if (meta.count[i] < meta.min) {
                min()[i]; } }
    } }
/* Key-value module */
symbolic int kv_items;
control kv(...) {....}
control NetCache( ... ) { /* NetCache module */
    apply {
        hash_inc.apply();
        find_min.apply();
        kv.apply(); } }
/* Utility function */
utility kv_sum = .004*kv_items+6.5
utility cms_error = -.00007*cols+.07
optimize 0.6*(kv_sum)+0.4*(1 - cms_error)
```

Figure 4: NetCache and Count-Min Sketch in P4All

Applications	P4 Code	P4All Code
NetCache	741	286
SketchLearn	366	88
Precision	283	266
ConQuest	694	649

Figure 5: P4All applications. The "P4 Code" and "P4All Code" columns show the lines of code of the original P4 application and the P4All implementation, respectively.

```
symbolic int cols;
```

Symbolic integers should be thought of as placeholders to be determined (and optimized for) at compile time. As in other general-purpose, solver-aided languages like Boogie [17], Sketch [29], or Rosette [30], the user leaves the choice of value up to P4All.

Often, programmers know constraints that are unknown to the compiler. For instance, user experience might suggest that count-min sketches with more than four hash functions offer diminishing returns (or simply might not be available). Such constraints may be written as assume statements:

```
assume 0 <= rows && rows < 4
```

An assume statement is related to the more familiar assert statement found in many conventional languages, such as C. However, an assert statement *fails* (causing program termination) when its underlying condition evaluates to false. An assume statement, in contrast, always *succeeds*, but adds constraints to the system, guaranteeing the execution can depend upon the conditions assumed.

3.2 Construct Elastic Data Structures

P4 data structures are defined using a combination of metadata and register arrays. The same is true of P4All. However, rather than using constants to define the extent of these structures, one uses symbolic values instead, so the compiler can optimize their extents.

In the count-min sketch, each row may be implemented as a register array (whose elements, in this case, are 32-bit integers to be used as counters). The size of each register array is the number of columns in a row. In P4All, we define this matrix as a symbolic array of register arrays:

```
register<bit<32>>(cols)[rows] cms;
```

In this declaration, we have a symbolic array `cms`, which contains `rows` instances of the register type. Each register array holds `cols` instances of 32-bit values.

Similarly, we can elastically define metadata fields. For a row in our CMS, we have an index and a count, each of which are 32-bit long fields, to store information from a single register in a row. Within the metadata struct, we define:

```
bit<32>[rows] index;
bit<32>[rows] count;
```

We declare two symbolic arrays (`index` and `count`) which contain `rows` instances of a 32-bit metadata field.

3.3 Define Elastic Operations in Loops

Because elastic data structures can stretch or contract to fit available resources, elastic operations over those structures must do more or less work accordingly. To accommodate such variation, P4All extends P4 with loops whose iteration count may be controlled by symbolic values.

Our CMS consists of two operations. The first operation hashes the input `rows` times, incrementing the result found in the CMS at that location, and puts the result in metadata. The second iterates over this metadata to compute the minimum at all hash locations. Operations are implemented using symbolic loops and are encapsulated in control blocks. The code below illustrates these operations.

```
/* actions used in control segments */
action incr()[int i] { ... }
action min()[int i] { ... }
/* hash and increment */
control hash_inc( ... ) {
  apply {
    for (i < rows) {
      incr()[i]; }
  } }
/* find global minimum */
control find_min( ... ) {
  apply {
    for (i < rows) {
```

```
      if (meta.count[i] < meta.min) {
        min()[i]; } }
  } }
```

These simple symbolic iterations (`for i < symbolic`) iterate from zero up to the symbolic bound, incrementing the index by one each time. The overarching NetCache algorithm can now call each control block in the ingress pipeline.

```
control NetCache( ... ) {
  apply {
    hash_inc.apply(...);
    find_min.apply(...); } }
```

Figure 5 lists four of the P4 applications that we translated into P4All using this methodology. We see reduction in lines of code with P4All because of symbolic loops that eliminate repeated code. We note that some applications saw a greater reduction than others, due to the use of C-like macros in the P4 code. Our discussions with P4 programmers suggest that while these macros make the program more compact, they make debugging more difficult. Additionally, changes to a program still require edits in multiple places, while P4All aims to make programs more robust by reducing the places a user must make edits.

3.4 Specify the Utility Function

Data structures designed for programmable switches are valid at a range of sizes. In the CMS example above, multiple assignments to `rows` and `cols` might fit within the resources of the switch. Finding the right parameters becomes even more difficult with multiple data structures involved. In the case of NetCache, after defining a CMS, the programmer still needs to define and optimize a key-value store.

To automate the process of selecting parameters, P4All allows programmers to define a linear, univariate utility function that expresses the relationship between the utility of a data structure and its size (as defined by symbolic values). Utility is a function of how well a data structure performs. For example, the error of a CMS and the hit rate of a key-value store are representations of utility for the respective data structures. The P4All compiler should find instances of the symbolic values that optimize this function, such that the resulting program fits within the switch resources.

The utility is often dependent on the workload, or the distribution of the network traffic. In this case, a user may want to include a workload-specific parameter in a utility function. We assume that the user knows what workload to expect before compiling and running their program. In order to tune the function for different distributions, the programmer can simply adjust this parameter.

For example, we can define the key-value store hit ratio as a function of its size for a workload with a Zipfian distribution. Suppose the key-value store has `kv_items` items. The probability of a request to the i^{th} most popular item is $\frac{1}{i^{\alpha}}$ [4]. In this case, α is a workload-dependent parameter that captures the amount of skew. Then, for `kv_items`, the probability of a cache hit is the sum of the probabilities for each item in the key-value store: $\sum_{i=1}^{kv\_items} \frac{1}{i^{\alpha}}$, divided by a function of the overall number of accesses. Because this function is not linear, we use an approximation as a utility function. We generate an approximation using NumPy's polyfit function [22], according to the above summation with $\alpha = 0.9$.

```
utility kv_sum = .004*kv_items+6.5
```

Similarly, we can define CMS error, ϵ, in terms of the number of columns, `cols`, in the sketch. For a workload α, we have $cols = 3(1/\epsilon)^{1/\alpha}$ [6]. While the number of rows in the CMS does not affect its error, and is thus not in the utility function, we can incorporate constraints in our compiler to guarantee a minimum number of rows. The number of rows, `rows`, in a CMS is used to determine a bound on the confidence, δ, of the estimations in the sketch ($rows = 2.5 \ln 1/\delta$) [6]. We use the following linear approximation, calculated using NumPy's polyfit function [22] for $\alpha = 0.9$:

```
utility cms_error = -.00007*cols+.07
```

In NetCache, the programmer must decide if either data structure should receive a higher proportion of the resources. If the CMS is prioritized, it can more accurately identify heavy hitters. However, the key-value store may not have sufficient space to store the frequently requested items. Conversely, if the CMS is too small, it cannot accurately measure which keys are popular and should be stored in the cache.

To capture the balance between data structures, a programmer can combine the utilities of each data structure into a weighted sum of the utilities. For the NetCache application, this means creating a utility function that slightly prioritizes the hit rate of the key-value store over the error of the CMS:

```
optimize 0.6*(kv_sum)+0.4*(1 - cms_error)
```

With a different workload or application, the utility function may change. In P4, this would require the user to rewrite their program, potentially introducing bugs or causing the code to fail to compile. However, with P4All, the programmer need only adjust the utility function, thus simplifying the "trial and error" process. This performance tuning occurs in a centralized place rather than being spread throughout the application, and hence, from a software engineering perspective, leads to a superior process.

Limitations. P4All relies on a user's ability to create a representative utility function and to assign weights to each portion of the utility function. Several of the data structures in Figure 1 have a well-defined notion of utility (e.g., CMS and KVS). However, this is not always true for each application, particularly those that combine data structures. We recognize that effectively designing a utility function can require significant effort to write these functions. We leave this problem as an open question.

In our examples, we have workload-dependent utility functions. Realistically, the programmer may not always know the workload ahead of time, or the workload may change over time. P4All, however, is a static system, meaning the user must manually adjust the workload parameter in their utility function. We leave as future work the creation of a dynamic system, which can change a utility function based on network measurements.

P4All optimizes a linear utility function, but the data structures we examine have more complex measures of utility. Our ongoing work includes extending P4All to support nonlinear, multivariate utility functions.

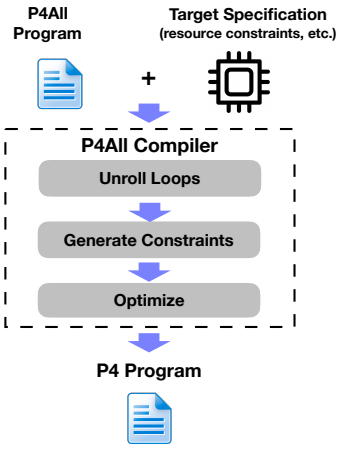

Figure 6: P4All compiler architecture.

4 P4ALL COMPILER ARCHITECTURE

The P4All compiler is responsible for assigning concrete values to each symbolic value, along with mapping program elements to a PISA switch. Figure 6 outlines a sketch of the compiler. The user provides a P4All program, along with a file specifying the resources of the target (e.g., number of stages, ALUs, etc. as defined by the switch designer), and the compiler uses these to create a resource-compliant mapping. While we use some of the existing P4 compilation methods to generate this mapping, the addition of symbolic values and loops presents new challenges in compiling to PISA hardware. In this section, we describe the steps of the compilation process and the challenges posed by symbolic values.

Unroll Loops. In P4 programs, there are a finite number of program elements (e.g., actions, registers, etc.). Hence, we must eliminate loops from P4All programs by unrolling them for a fixed number of iterations. We then need to know the value of the symbolic value that controls the loop. However, in P4All compilation, we do not know the optimal number of loop iterations until after the compilation process. We cannot unroll symbolic loops without concrete values, but we cannot determine optimal concrete values without unrolling loops. To solve this circularity, our compiler will first determine upper bounds for symbolic loops. One way the compiler can handle this is by analyzing dependencies between actions, since dependent actions must occur in separate stages.

Generate Constraints. As a second step, the P4All compiler generates constraints for optimization. These represent target-specific resource constraints of the switch, as given by the target specification file. They ensure the compiler's output respects the target resources. For example, we have a constraint which prevents the solution from using more memory than what is available. A P4All compiler can use the constraints in existing P4 compilers (e.g., [16]).

Optimize. Lastly, the P4All compiler generates and solves an optimization problem that maximizes a linear utility function. The solution to this problem is concrete values for each of the symbolic values and a mapping of program elements to pipeline stages.

Compiler Output. In our initial experiments, the P4All compiler generates a P4 file that is then compiled with the Barefoot Tofino compiler. A switch vendor could also incorporate the P4All compiler into their P4 compiler to create a single, unified compiler.

While we envision P4All as part of existing P4 compilers, we recognize that some target-specific constraints may be difficult to model. For this reason, initially our compiler operates as a preprocessor and generates a P4 file. Because we do not model every target-specific constraint, our compiler could generate a solution that does not compile to a physical switch. Then, we must recompile the P4All program with additional constraints to limit the solution.

5 RELATED WORK

Languages for network programming. While P4 makes it possible to create exciting new applications over a variety of hardware targets, it does not make it easy. Domino [27] and Chipmunk [7] utilize a high-level C-like language to aid in programming packet-processing algorithms. P4All also aims to simplify this programming process, but by enhancing P4 with elastic data structures via symbolic values and loops. Whereas these systems optimize data-plane layout for static, fixed-sized data structures, P4All optimizes the structure itself to make the most effective use of resources.

Using synthesis for compiling to PISA. Domino [27] is a programming language for writing data-plane algorithms in a C-like syntax. The Domino compiler extracts "codelets", groups of statements that must execute simultaneously. It uses program synthesis (SKETCH [29]) to map a codelet to ALUs (atoms in the paper's terminology) in each stage. If codelets violate target constraints, the program is rejected. Chipmunk [7] uses syntax-guided synthesis to perform an exhaustive search of all mappings of the program to a target. Thus, it can find mappings that might be missed by Domino. Both Domino and Chipmunk map programs with fixed-size data structures, while P4All enables elastic structures.

Compiling to RMT. Jose et al. [16] use ILPs and greedy algorithms to compile packet-processing programs for RMT [3] and FlexPipe [23] architectures. The P4All compiler can add additional constraints for resolving symbolic values to these approaches.

6 CONCLUSION AND FUTURE WORK

This paper introduces the concept of *elastic switch programs*—programs that contain data structures capable of expanding to use the resources available on a particular hardware target. Compared to their inelastic counterparts, elastic programs are more modular (as they can stretch or contract depending on the resource needs of other components on the switch) and more portable (as they can be recompiled to a range of different targets). We believe that the P4All language and our reusable modules will make it easier to implement and deploy a range of future data-plane applications.

In addition to the open questions discussed throughout the paper, we believe an object-oriented programming model would greatly promote reusability in P4All. We envision objects such as the data structures in Figure 1 being reused in a variety of applications.

ACKNOWLEDGMENTS

This work is supported by DARPA under Dispersed Computing HR0011-17-C-0047 and NSF under FMiTF-1837030 and CNS-1703493.

REFERENCES

[1] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *IEEE International Conference on Network Protocols*. 313–323.

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[3] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM*. 99–110.

[4] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1999. Web Caching and Zipf-like Distributions: Evidence and Implications. In *IEEE INFOCOM*. 126–134.

[5] Graham Cormode and S. Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-min Sketch and Its Applications. *Journal of Algorithms* 55, 1 (April 2005), 58–75. https://doi.org/10.1016/j.jalgor.2003.12.001

[6] Graham Cormode and S. Muthukrishnan. 2005. Summarizing and Mining Skewed Data Streams. In *SIAM International Conference on Data Mining*, Hillol Kargupta, Jaideep Srivastava, Chandrika Kamath, and Arnold Goodman (Eds.). SIAM, 44–55. https://doi.org/10.1137/1.9781611972757.5

[7] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. 2019. Autogenerating Fast Packet-Processing Code Using Program Synthesis. In *ACM SIGCOMM Workshop on Hot Topics in Networks*. 150–160. https://doi.org/10.1145/3365609.3365858

[8] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven Streaming Network Telemetry. In *ACM SIGCOMM*. ACM, 357–371. https://doi.org/10.1145/3230543.3230555

[9] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. 2018. Network-Wide Heavy Hitter Detection with Commodity Switches. In *Symposium on SDN Research*. Article Article 8, 7 pages.

[10] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast Connectivity Recovery Entirely in the Data Plane. In *USENIX Symposium on Networked Systems Design and Implementation*. Boston, MA, 161–176. https://www.usenix.org/conference/nsdi19/presentation/holterbach

[11] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. SketchVisor: Robust Network Measurement for Software Packet Processing. In *ACM SIGCOMM*. 113–126.

[12] Qun Huang, Patrick P. C. Lee, and Yungang Bao. 2018. SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. In *ACM SIGCOMM*. 576–590. https://doi.org/10.1145/3230543.3230559

[13] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. 2018. Life in the Fast Lane: A Line-Rate Linear Road. In *Symposium on SDN Research*.

[14] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX Symposium on Networked Systems Design and Implementation*. Renton, WA, 35–49. https://www.usenix.org/conference/nsdi18/presentation/jin

[15] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soule, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Symposium on Operating System Principles*.

[16] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. 2015. Compiling Packet Programs to Reconfigurable Switches. In *USENIX Conference on Networked Systems Design and Implementation*. USA, 103–115.

[17] K. Rustan M. Leino and Philipp Rümmer. 2010. A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In *Tools and Algorithms for the Construction and Analysis of Systems*, Javier Esparza and Rupak Majumdar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 312–327.

[18] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *USENIX Symposium on Networked Systems Design and Implementation*. Santa Clara, CA, 311–324. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/li-yuliang

[19] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. NitroSketch: Robust and General Sketch-Based Monitoring in Software Switches. In *ACM SIGCOMM*. 334–350. https://doi.org/10.1145/3341302.3342076

[20] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *ACM SIGCOMM*. 101–114. https://doi.org/10.1145/2934872.2934906

[21] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *ACM SIGCOMM*. 15–28.

[22] Travis E Oliphant. 2006. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA.

[23] Recep Ozdag. 2012. Intel® Ethernet Switch FM6000 Series–Software Defined Networking. goo.gl/AnvOvX.

[24] P4 Language Consortium. 2018. $P4_{16}$ Language Specifications. (2018). https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf

[25] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. 2017. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *USENIX Networked Systems Design and Implementation*. 67–82. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/sharma

[26] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *USENIX Symposium on Networked Systems Design and Implementation*. Renton, WA, 1–16. https://www.usenix.org/conference/nsdi18/presentation/sharma

[27] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *ACM SIGCOMM*. 15–28.

[28] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *ACM SIGCOMM Symposium on SDN Research*. ACM, 164–176. https://doi.org/10.1145/3050220.3063772

[29] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Architectural Support for Programming Languages and Operating Systems*. 404–415.

[30] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 530–541. https://doi.org/10.1145/2594291.2594340

[31] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *ACM SIGCOMM*. ACM, 561–575. https://doi.org/10.1145/3230543.3230544