Modular Switch Programming Under Resource Constraints

Mary Hogan[1], Shir Landau-Feibish[2], Mina Tahmasbi Arashloo[3], Jennifer Rexford[1], and David Walker[1]

[1]Princeton University
[2]The Open University of Israel
[3]Cornell University

Abstract

Programmable networks support a wide variety of applications, including access control, routing, monitoring, caching, and synchronization. As demand for applications grows, so does resource contention within the switch data plane. Cramming applications onto a switch is a challenging task that often results in non-modular programming, frustrating "trial and error" compile-debug cycles, and suboptimal use of resources. In this paper, we present P4All, an extension of P4 that allows programmers to define *elastic* data structures that stretch automatically to make optimal use of available switch resources. These data structures are defined using *symbolic primitives* (that parameterize the size and shape of the structure) and *objective functions* (that quantify the value gained or lost as that shape changes). A top-level optimization function specifies how to share resources amongst data structures or applications. We demonstrate the inherent modularity and effectiveness of our design by building a range of reusable elastic data structures including hash tables, Bloom filters, sketches, and key-value stores, and using those structures within larger applications. We show how to implement the P4All compiler using a combination of dependency analysis, loop unrolling, linear and non-linear constraint generation, and constraint solving. We evaluate the compiler's performance, showing that a range of elastic programs can be compiled to P4 in few minutes at most, but usually less.

1 Introduction

P4 has quickly become a key language for programming network data planes. Using P4, operators can define their own packet headers and specify how the data plane should parse and process them [7]. In addition to implementing traditional forwarding, routing, and load-balancing tasks, this flexibility has enabled new kinds of in-network computing that can accelerate distributed applications [26, 27] and perform advanced monitoring and telemetry [10, 11, 17, 30].

All of these applications place demands on switch resources, but for many, the demands are somewhat flexible: additional resources, typically memory or stages in the PISA pipeline, improve application performance, but do not necessarily make or break it. For instance, NetCache [27] improves throughput and latency for key-value stores via in-network computing. Internally, it uses two main data structures: a count-min sketch (CMS) to keep track of popular keys, and a compact key-value store (KVS) to maintain their corresponding values. Increasing or decreasing the size of those structures will have an impact on performance, but does not affect the correctness of the system—a cache miss may increase latency, but the correct values will always be returned for a given key. Other applications, such as traffic-monitoring infrastructure, have similar properties. Increasing the size of the underlying hash tables, Bloom filters, sketches, or key-value stores may make network monitoring somewhat more precise but does not typically result in all-or-nothing decisions.

Because resource constraints for these components are flexible, network engineers can, in theory, squeeze multiple different applications onto a single device. Unfortunately, however, doing so using today's programming language technology is a challenging and error-prone task: P4 forces programmers to hardcode their decisions about the size and shape of their data structures. If the data structure is too large, the program simply fails to compile and little feedback is provided; if it is too small, it will compile but the resources will be used suboptimally. Moreover, structures are not reuseable: a cache, that fits just fine on a switch alongside a table for IP forwarding, is suddenly too large when a firewall is added. To squeeze the cache in, programmers may have to rewrite the internals of their cache, manually adjusting the number or sizes of the registers or match-action tables used. To test their work, they resort to a tedious trial-and-error cycle of rewriting their applications, and invoking the compiler to see if it can succeed in fitting the structures into the available hardware resources.

This manual process of tweaking the *internal* details of data structures, and checking whether the resulting structures satisfy *global* constraints, is inherently non-modular: Programmers tasked with implementing separate applications cannot do so independently. Indeed, while the same data structures

Data Structure	Used in
Key-value store/ hash table	Precision [6], Sonata [17], Network-Wide HH [19], Carpe [20], Sketchvisor [23], LinearRoad [25], NetChain [26], NetCache [27], FlowRadar [30], HashPipe [41], Elastic Sketch [46]
Hash-based matrix (Sketch)	AROMA [4], Sketchvisor [23], Sketchlearn [24], NetCache [27], Nitrosketch [31], UnivMon [32], Sharma et al. [38], Fair Queueing [39], Elastic Sketch [46]
Bloom filter	NetCache [27], FlowRadar [30], SilkRoad [34], Sharma et al. [38]
Multi-value table	BeauCoup [10], Blink [22]
Sliding window sketch	PINT [5], Conquest [11]
Ring buffer	NetLock [47], Netseer [48]

Figure 1: PISA data structures

appear again and again (see Figure 1 for a selection), the varying resource constraints makes it difficult to reuse these structures for different targets or applications.

Elastic Switch Programming. We extend P4 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 or "contract" to squeeze in beside other applications. Elastic programs can be constructed from any number of elastic components that each stretch arbitrarily to fill available space. An elastic NetCache program, for example, may be constructed from an elastic count-min sketch and an elastic key-value store. The programmer can control the relative stretch of these modules by specifying an objective function that the compiler should maximize. For example, the NetCache application could maximize the cache "hit rate" by prioritizing memory allocation 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 processing units and pipeline stages.

To implement these elastic programs, we present P4All, a backward-compatible extension of the P4 language with several additional features: (1) symbolic values, (2) symbolic arrays, (3) bounded loops with iteration counts governed by symbolic values, (4) local objective functions for data structures, and (5) global optimization criteria. Symbolic values make the sizes of arrays and other state flexible, allowing them to stretch as needed. Loops indexed by symbolic values make it possible to construct operations over elastic data structures. Objective functions provide a principled way for the programmer to describe the relative gain/loss from growing/shrinking individual data structures. Global optimization criteria make it possible to weight the relative importance of each structure or application residing on a shared device.

We have implemented a compiler for P4All that operates

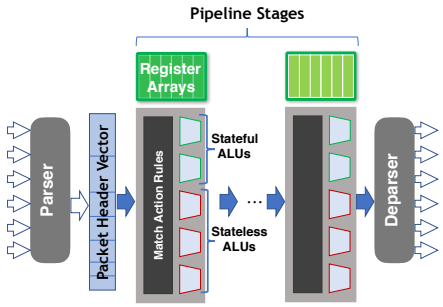


Figure 2: Protocol Independent Switch Architecture (PISA)

in two main stages. First, it computes an upper bound on the number of possible iterations of loops, so it can produce a simpler optimization problem over unrolled, loop-free code. This upper bound is computed by conservatively analyzing the dependency structure of the loop bodies and their resource utilization. Next, the compiler unrolls the loops to those bounds and generates a constraint system that optimizes the resource utilization of the loop-free code for a particular target. We use the Intel Tofino chip as our target. We evaluate our system by developing a number of reusable, elastic structures and building several elastic applications using these structures. Our experiments show that the P4All compiler runs in a matter of minutes (or less) and produces P4 programs that are competitive with hand-optimized code. This paper builds on our earlier workshop paper [21] by extending the language for nonlinear objective functions over multiple variables. We also implement the optimization problem and compiler outlined in the workshop paper, along with evaluating it with a variety of data structures.

In summary, we make the following contributions.

- The design of P4All, a backward-compatible extension to P4 that enables elastic network programming.
- The implementation of an optimizing compiler for P4All.
- A library of reusable elastic data structures, including their objective functions, and examples of combining them to create sophisticated applications.
- An evaluation of our system on a range of applications.

2 P4 Programming Challenges

Programming PISA devices is difficult because the resources available are limited and partitioned across pipeline stages. The architecture forces programmers to keep track of implicit dependencies between actions, lay out those actions across stages, compute memory requirements of each task, and fit the jigsaw pieces emerging from many independent tasks together into the overall resource-constrained puzzle of the pipeline.

2.1 Constrained Data-Plane Resources

P4 is designed to program a *Protocol Independent Switch Architecture* (PISA) data plane (Figure 2). Such an architecture contains a programmable packet parser, processing pipeline, and deparser. When a packet enters the switch, the parser extracts information from the packet and populates the *Packet Header Vector* (PHV). The PHV contains information from the packet's various fields, such as the source IP, TCP port, *etc.* that are relevant to the switch's task, whether it be routing, monitoring, or load balancing. The PHV also stores additional per-packet data, or *metadata*. Metadata often holds temporary values or intermediate results required by the application. Finally, the deparser reverses the function of the parser, using the PHV to reconstitute a packet and send it on its way.

Between parser and deparser sits a packet-processing pipeline. A program may recirculate a packet by sending it back to the beginning, but too much recirculation decreases throughput. Each stage contains a fixed set of resources.

- **Pipeline stages.** The processing pipeline is composed of a fixed number (S) of stages.
- **Packet header vector (PHV).** The PHV that carries information from packet fields and additional per-packet metadata through the pipeline has limited width (P bits).
- **Registers.** A stage is associated with M bits of registers (of limited width) that serve as persistent memory.
- **Match-action rules.** Each stage stores match-action rules in either TCAM or SRAM (T bits).
- **ALUs.** Actions are performed by ALUs associated with a stage. Each stage has F stateful ALUs (that perform actions requiring registers) and L stateless ALUs (that do not).
- **Hash units.** Each stage can perform N hashes at once.

The P4 language helps manage data-plane resources by providing a layer of abstraction above PISA. A P4 compiler maps these higher-level abstractions down to the PISA architecture and organizes the computation into stages. However, experience with programming in P4 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 PISA switch. To do this well, programmers must recognize dependencies between actions, estimate the stages available and consider the memory layout and usage of their programs—in short, they must redo many of the jobs of the compiler. These are difficult jobs to do well, even for world-experts, and next to impossible for novices. Inevitably, attempts at estimating resource bounds leads to some amount of trial and error. 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.

2.2 Example: Implementing NetCache in P4

To illustrate some of the difficulties of programming with P4, consider an engineer in charge of upgrading their network to include a new caching subsystem, based on NetCache [27], which is designed to accelerate response times for web services. NetCache contains two main data structures, a *count-min sketch* (CMS) for keeping track of the popularity of the keys, and a *key-value store* (KVS) to map popular keys to values. Like any good programmer, our engineer constructs these two data structures modularly, one at a time.

First, the engineer implements 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, the 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 ($\{h_i\}$), one for each row, that range from $(1 \ldots w)$. In each row, the output of the hash function determines which column in the row is incremented for f_x. For example, in the second row of the CMS, hash function h_2 determines that column ($h_2(f_x)$) is incremented. 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.

The CMS may overestimate the number of occurrences of a packet x if there are hash collisions. Increasing the size of the sketch in any dimension—either by adding more rows (*i.e.*, additional, different hash functions) or by increasing the range of the hash functions—can improve accuracy. Our engineer must decide how to assign resources to the CMS, including how much memory to allocate and how to divide memory into rows. This allocation becomes even harder when grappling with dividing resources between multiple structures.

Figure 3 presents a fragment of a P4 program that implements a CMS. Lines 1-7 declare the metadata used by the CMS to store a count at a particular index (a hash of a flow id). Lines 10-12 declare the low-level data structures (registers) that actually make up the CMS—four rows ($r = 4$) of columns ($w = 2048$) that can each store values represented by 32 bits. Lines 14-16 and 18-20 declare the actions for hashing/incrementing and for updating the metadata designed to store the global minimum. Both actions use metadata, another constrained resource that must be accounted for. The hashing action is a complex action containing several atomic actions: (1) an action to hash the key to an index into 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. Such multi-part actions can demand a number of resources, including several ALUs. As our engineer adds more of these actions to the program, it becomes increasingly difficult to estimate the resource requirements. In the *apply* fragment of the P4 program (lines 22-30), the program first executes all the hash actions, computing and storing counts for each hash function, and then compares those counts

```
struct custom_metadata_t {
    bit<32> min;
    bit<32> index0;
    bit<32> count0;
    ...
    bit<32> index3;
    bit<32> count3; }
control Ingress( ... ) {
    /* a register array for each hash table */
    register<bit<32>>(2048) counter0;
    ...
    register<bit<32>>(2048) counter3;
   /* an action to update each hash table */
    action incr_0() { ... }
    ...
    action incr_3() { ... }
    /* an action to set the minimum */
    action min_0(){meta.min = meta.count0;}
    ...
    action min_3(){ . . . }
    /* execute the following on each packet */
    apply {
        meta.min = 0; /*initialize global min*/
        /* compute hashes */
        incr_0(); ... incr_3();
        /* compute minimum */
        if (meta.count0 < meta.min) { min_0();}
        ...
        if (meta.count3 < meta.min) { min_3();}
        } }
```

Figure 3: Count-Min Sketch in $P4_{16}$

to each other looking for the minimal one.

Upon reviewing this code, some of the deficiencies of P4 should immediately be apparent. First, there is a great deal of repeated code: Repeated data-structure definitions, action definitions, and invocations of those action definitions in the apply segment of the program. Good programming languages make it possible to avoid repeated code by allowing programmers to craft reusable abstractions. Avoiding repetition in programming has all sorts of good properties including the fact that when errors occur or when changes need to be made, they only need to be fixed/made in one place. 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 the programmer had to choose magic constants (like 2048) and test whether such constants lead to programs that can be compiled or not.

3 Elastic Programming in P4All

P4All improves upon P4 by making it possible to construct and manipulate *elastic data structures*. These data structures may be developed modularly and combined, off-the-shelf, to build efficient new applications. In this section, we illustrate language features by building an elastic count-min sketch and using it in the NetCache application (see also Figure 4).

```
/* Count-min sketch module */
symbolic rows;
symbolic cols;
assume cols > 0;
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]; } } } }
objective cms_obj {
    function: scale (3.0/cols);
    step: 100; }

/* Key-value module */
symbolic k;  /* number of items */
assume k > 0;
control kv(...) {....}
/* NetCache module */
control NetCache( ... ) {
    apply {
        hash_inc.apply();
        find_min.apply();
        kv.apply(); } }
objective kvs_obj {
    function: scale (sum(map(lambda y: 1.0/
        y,range(1,k+1))));
    step: 100; }
maximize 0.8*kvs_obj-0.2*cms_obj
```

Figure 4: NetCache and Count-Min Sketch in P4All

3.1 Declare the Elastic Parameters

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

```
symbolic rows;
symbolic cols;
```

Symbolic integers like `rows` and `cols` should be thought of as "some integer"—they are placeholders that are determined (and optimized for) at compile time. In other words, as in other general-purpose, solver-aided languages like Boogie [29], Sketch [42], or Rosette [43], the programmer leaves the choice of value up to the P4All compiler.

Often, programmers know constraints that are unknown to the compiler. For instance, programmer experience might suggest that count-min sketches with more than four hash

functions offer diminishing returns. Such constraints may be written as assume statements as follows:

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

An assume statement is related to the familiar assert statement found in languages like 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 Declare Elastic State

The next step in defining an elastic data structure is to declare elastic state. P4 data structures are defined using a combination of the packet-header vector (metadata associated with each packet), registers (updated within the data plane), or match-action tables (rules installed by the control plane). The same is true of P4All. However, rather than using constants to define the extent of the state, one uses symbolic values, so the compiler can optimize their extents for the programmer.

In the count-min sketch, each row may be implemented as a register array (whose elements, in this case, are 32-bit integers used as counters). The number of registers in 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.

One can also define elastic metadata. For instance, for each row of the CMS, we need metadata to record an index and count for that row. To do so, we define symbolic arrays of metadata as follows. Each element of each array is a 32-bit field. The arrays each contain `rows` items.

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

3.3 Define Elastic Operations

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

The count-min sketch of our running example consists of two operations. The first operation hashes the input `rows` times, incrementing the result found in the CMS at that location, and storing the result in the metadata. The second iterates over this metadata to compute the overall minimum found at all hash locations. Each operation is implemented using symbolic loops and is encapsulated in its own control block. 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 < rows`) iterate from zero up to the symbolic bound (`rows`), 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(...);
    ... } }
```

3.4 Specify the Objective Function

Data structures written for programmable switches are valid for 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 harder when a program has multiple data structures. 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 an objective function that expresses the relationship between the utility of the structure and its size (as defined by symbolic values). For example, the CMS gains utility as one increases the cols parameter, because CMS error rate decreases. The P4All compiler should find instances of the symbolic values that optimize the given user-defined function subject to the constraint that the resulting program can fit within the switch resources.

For example, we can define the hit ratio for the key-value store as a function of its size for a workload with a Zipfian distribution. Suppose the key-value store has k items. The probability of a request to the i^{th} most popular item is $\frac{1}{i^{\alpha}}$ [9]. In this case, α is a workload-dependent parameter that captures the amount of skew in the distribution. Then, for k items, the probability of a cache hit is the sum of the probabilities for each item in the key-value store: $\sum_{i=1}^{k} \frac{1}{i^{\alpha}}$. Hence, in P4All, for $\alpha = 1$, we might define the following objective function.

```
sum(map(lambda y: 1.0/y,range(1,k+1)))
```

In practice, we have found that non-linear optimization functions that use division can generate poor quality solutions, perhaps due to rounding errors (at least for the solver,

Gurobi [18], that we use). Hence, we *scale* such functions up, which results in the following optimization function.

```
scale(sum(map(lambda y: 1.0/y,range(1,k+1))))
```

Because we supply programmers with a library of reusable structures and optimization functions for them, non-expert programmers who use our libraries do not have to concern themselves with such details.

Similarly, we can define CMS error, ε, in terms of the number of columns, w, in the sketch. For a workload with parameter α, we can set $w = 3(1/\varepsilon)^{1/\alpha}$ [13]. The number of rows in the CMS does not affect ε, so we may choose to leave it out of the objective function. However, we can incorporate constraints to guarantee a minimum number of rows. The number of rows, d, in a CMS is used to determine a bound on the confidence, δ, of the estimations in the sketch ($d = 2.5 \ln 1/\delta$) [13]. For $\alpha = 1$, this objective function is `3.0/cols`.

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 objectives of each data structure into a weighted sum. For the NetCache application, this means creating an objective function that slightly prioritizes the hit rate of the key-value store over the error of the CMS:

```
maximize 0.8*kvs_obj-0.2*cms_obj
```

Figure 5 presents the symbolic values and possible objective functions for different data structures. Each structure has symbolic values and an objective function derived from the purpose of the structure, which may vary across applications. For example, the key-value store used in NetCache [27] acts as a cache, and the main goal of the algorithm is to maximize the cache hits. In the case of a collision in the hash table used in BeauCoup [10], only one of the values is kept , and the other is discarded, resulting in possible errors. Therefore, the main goal of the algorithm is to minimize collisions. The programmer can define the objective function of each structure based on the specific needs of the system. Existing analyses of common data structures can assist in defining these functions. For example, for the Bloom filter, the probability for false positives in Zipfian-distributed traffic has been analyzed by Cohen and Matias [12].

Complex Objectives. Some objective functions (*e.g.*, CMS) may only include a single symbolic variable, while others are a function of multiple variables (*e.g.*, Bloom filter in Figure 5). Because our compiler uses Gurobi [18] in the back end to solve optimization problems, it is bound by Gurobi's constraints. In particular, Gurobi cannot solve complex, non-linear objectives that are functions of multiple variables directly. As a consequence, we tackle these objectives in two steps. First, we transform objectives in multiple variables (say, x and y) into objectives in a single variable (say x), by choosing a set of possible values of y to consider. We create a different Gurobi instance for each value of y, solve all the instances independently (a highly parallelizable task) and find the global optimum afterwards. Second, we use Gurobi to implement piece-wise linear approximations of the non-linear functions. Both of these steps benefit from some user input, and we have extended P4All to accommodate such input.

To reduce objectives with multiple variables to a single variable, we allow users to provide a set of points at which to consider evaluating certain symbolic values. Doing so provides users some control over the number of Gurobi instances generated and hence the compilation costs of solving complex optimization problems. Such sets can be generated via "range notation" (optionally including a stride, not shown here). For example, a possible objective function for a Bloom filter depends on the number of bits in the filter as well as the number of hash functions used. To eliminate the second variable from the subsequent optimization objective, a programmer can define the symbolic variable `hashes` as follows.

```
symbolic hashes [1..10]
```

On processing such a declaration, the compiler generates ten separate optimization problems, one for each potential value of the hash functions. The compiler chooses the solution from the instance that generated the optimal objective, and it outputs the program layout and the concrete values for the number of hashes and number of bits in the filter.

To reduce non-linear functions to linear ones, piecewise linear approximations are used. By default, the compiler will use the simplest such approximation: a single line. Doing so results in fast compile times, but can lead to suboptimal solutions. To improve the quality of solution, we allow programmers to specify the number of linear pieces using a "step" annotation on their objective function. For instance, on lines 21-23 of Figure 4, the objective for the CMS is defined with a simple function and a "step" of 100, indicating that a linear component is created between every 100th value. Increasing the number of linear components in the approximation can increase the cost of solving these optimization problems. By providing programmers with optional control, we support a "pay-as-you-go" model that allows programmers to trade compile time for precision if they so choose.

4 Compiling Elastic Programs

Inputs to the P4All compiler include a P4All program and a specification of the target's resources (*i.e.*, the PISA resource parameters defined Section 2.1 and the capabilities of the ALUs). The compiler outputs a P4 program with a concrete

Module	Symbolic values	Intuition	Objective Function
Key-value store/ hash table	Number of rows k	NetCache [27]: Maximize cache hits	maximize $\sum_{i=1}^{k} \frac{1}{i^{\alpha}}$
Hash-based matrix (Sketch)	Num rows d, num columns w	NetCache [27] (CMS): Minimize heavy hitter detection error	minimize $\varepsilon = (\frac{3}{w})^{\alpha}$
Bloom filter	Num bits m, num hash functions k	NetCache [27]: Minimize false positives	minimize $(1-e^{-\alpha}(1+\frac{\alpha m}{m-1}))^{k}$
Multi-value table	Number of rows k	BeauCoup [10]: Minimize collisions. BeauCoup parameter set B; Probability to insert to table p = $f(\alpha, B)$; Expected number of items in stream n	minimize $(\frac{1}{k})^{n \cdot p}$
Sliding window sketch	Num rows d, num columns w, num epochs t	ConQuest [11]: Maximize epochs and minimize error	maximize $t(1-(\frac{3}{w})^{\alpha})$
Ring buffer	Buffer length b	Netseer [48]: Maximize buffer capacity	maximize b

Figure 5: Symbolic values and objective functions for Zipfian distributed traffic with (constant) parameter α.

assignment for each symbolic value, and a mapping of P4 program elements to stages in the target's pipeline. The output program is a *valid instance* of the input when the concrete values chosen to replace symbolic ones satisfy the user constraints (*i.e.*, `assume` statements) as well as the constraints of the PISA model that is targeted. In addition, loops are unrolled as indicated given the chosen concrete values. The output program is an *optimal instance*, when in addition to being valid, it maximizes the given objective function.

The P4All compiler first analyzes the control and data dependencies between actions in the program to compute an *upper bound* on the number of times each loop can be unrolled without exhausting the target's resources (§4.1). For example, a for-loop with a dependency across successive iterations cannot run more times than the number of pipeline stages (S). The unrolled program also cannot require more ALUs than exist on the target ($(F+L)*S$).

Next, the compiler generates an integer linear program (ILP) with variables and constraints that govern the quantity and placement of actions, registers, and metadata relative to the target constraints (§4.2). The upper bound ensures this integer linear program is "large enough" to consider all possible placements of program elements that can maximize the use of resources. However, the ILP is more accurate than the coarse unrolling approximation we use. Hence, it may generate a solution that excludes some of the unrolled iterations—some of the later iterations may ultimately not "fit" in the data plane or may not optimize the user's preferred objective function when other constraints are accounted for. The resulting ILP solution is a layout of the program on the target, including the stage placement and memory allocation, and optimal concrete assignments for the symbolic values. Throughout this section, we use the CMS program in Figure 4 as a running example. For the sake of the example, we assume that the target has three pipeline stages ($S = 3$), 2048b memory per stage ($M = 2048$), two stateful and two stateless ALUs per stage ($F = L = 2$), and 4096 bits of PHV ($P = 4096$).

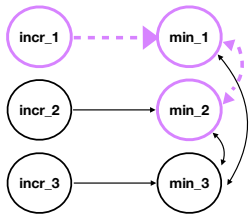


Figure 6: An example dependency graph used for computing upper bounds for loop unrolling (§4.1).

4.1 Upper Bounds for Loop Unrolling

In its first stage, the P4All compiler finds upper bounds for symbolic values bounding the input program's loops. To find an upper bound for a symbolic value v governing the number of iterations of some loop, the compiler first identifies all of the loops bounded by v. It then generates a graph G_v that captures the dependencies between the actions in each iteration of each loop and between successive iterations. It uses the information represented in G_v and the target's resource constraints to compute the upper bound.

Determining dependencies. When a loop is unrolled K times, it is replaced by K repetitions of the code in its body such that in repetition i, each action a in the original body of the loop is renamed to a_i. The compiler constructs the dependency graph G_v based on the actions in the unrolled bodies of for-loops bounded by v. Each node n in the dependency graph G_v represents a set A_n of actions that access the same register and thus *must* be placed in the same stage.

Dependency graphs can have (1) *precedence edges*, which are one-way, directed edges, and (2) *exclusion edges*, which are bidirectional. There is a precedence edge from node n_1 to node n_2 (indicated with the notation $n_1 \longrightarrow n_2$) if there is a data or control dependency from any of the actions represented by n_1 to any of the actions represented by n_2. The presence of the edge $n_1 \longrightarrow n_2$ forces all actions associated with n_1 to be placed in a stage that strictly precedes the stage where actions of n_2 are placed. In contrast, an exclusion edge ($n_1 \longleftrightarrow n_2$) indicates the actions of n_1 must be placed in a separate stage from the actions of n_2 but n_1 need not precede

n_2. In general, when actions are commutative, but cannot share a stage, they will be separated by exclusion edges. For instance, if actions a_1 and and a_2 both add one to the same metadata field, they cannot be placed in the same stage, but they commute: a_1 may precede a_2 or a_2 may precede a_1.

Figure 6 shows the dependency graph for `rows` from our CMS example. Only the `incr_i` actions access register arrays, and they all access different arrays. Thus, each node represents only one action. There is a precedence edge from `incr_i` to `min_i` as the former writes to the same metadata variable read by the latter. Thus, `incr_i` must be placed in a stage preceding `min_i`. There are exclusion edges between each pair of `min_i` and `min_j` because they are commutative but write to the same metadata fields: `min_i` sets the metadata variable tracking the global minimum `meta.min` to the minimum of its current value and the ith row of the CMS (`meta.count[i]`).

Computing the upper bound. To compute an upper bound for loops guarded by v, our compiler unrolls for-loops bounded by v for increasing values of K, generating a graph G_v until one of the following two criteria are satisfied:

1. the length of the longest simple path in G_v exceeds the total number of stages S, or
2. the total number of ALUs required to implement actions across all nodes in G_v exceeds the total number of ALUs on the target (*i.e.*, $(F+L)*S$).

Once either of the above criteria are satisfied, the compiler can use the current value of K, *i.e.*, the number of times the loops have been unrolled, as an upper bound for v. This is because any simple path in G_v represents a sequence of actions that must be laid out in disjoint stages. Hence, a simple path longer than the total number of stages cannot be implemented on the switch (*i.e.*, criteria 1). Likewise, the switch has only $(F+L)*S$ ALUs and a computation that requires more cannot be implemented (*i.e.*, criteria 2).

Figure 6 presents an analysis of a CMS loop bounded by `rows`. Notice that the length of the longest simple path in G_{rows} will exceed the number of stages ($S=3$) when three iterations of the loop have been unrolled. On the other hand, when only two iterations of the loop are unrolled, the longest simple path has length 3 and will fit. Thus, the compiler computes 2 as the upper bound for this loop.

Nested loops. To manage nested loops, we apply the algorithm described above to each loop, making the most conservative assumption about the other loops. For instance, suppose the program has a loop with nesting depth 2 in which the outer loop bounded by v_{out} and the inner loop is bounded by v_{in}. Assume also the valid range of values for both v_{in} and v_{out} is $(1,\infty]$. The compiler sets v_{in} to one, unrolls the inner loop, and computes an upper bound for v_{out} as described above. Next, the compiler sets v_{out} to one, unrolls the outer loop, and proceeds to compute the upper bound for v_{in} as described above. In theory, heavily nested loops could lead to an explosion in the complexity of our algorithm, but in practice, we have not found nested loops common or problematic. Only our SketchLearn application requires nested loops and the nesting depth is just 2, which is easily handled by our system.

Variables		
Actions	#1	$\{x_{a_i,s} \mid 0 \le s < S\}$
Registers	#2	$\{m_{r_i,s} \mid 0 \le s < S\}$
Match-Action Tables	#3	$\{tm_{t_i,s} \mid 0 \le s < S\}$
Metadata	#4	$\{d_i \mid i \le U_v\}$
Constraints		
Dependencies		
Same-Stage	#5	$x_{a_i,s} = x_{b_i,s} s < S$
Exclusion	#6	$x_{a_i,s} \le 1 - x_{b_i,s}$ $s < S$
Precedence	#7	$x_{b_i,y} \le 1 - x_{a_i,z}$ $y,z < S, y \le z$
Conditional	#8	$\sum_{0\le s<S} x_{a_i,s} = \sum_{0\le s<S} x_{b_i,s}$ $0 \le i \le U_v$
Resources		
Memory	#9	$\sum_i m_{r_i,s} \cdot w_{r_i} \le M \quad \forall s < S$
	#10	$m_{r_i,s} \le x_{a_i,s} \cdot M \quad 0 \le s < S$
	#11	$m_{r_i,s} \cdot w_0 = m_{0,s} \cdot w_{r_i}$ $\forall s < S, r \ge 1$
TCAM	#12	$\sum_i tm_{t_i,s} \cdot tw_{t_i} \le T \quad \forall s < S$
Stateful ALUs	#13	$\sum_i H_f(a_i) \cdot x_{a_i,s} \le F$ $\forall 0 \le s < S$
Stateless ALUs	#14	$\sum_i H_l(a_i) \cdot x_{a_i,s} \le L$ $\forall 0 \le s < S$
PHV	#15	$\sum_i d_i \cdot bits_d \le P - P_{fixed}$
	#16	$d_i = \sum_{0\le s<S} x_{a_i,s}$ if accesses(a,d)
Hash Functions	#17	$\sum_i h_{ha_i,s} \le N \quad \forall s < S$
Others		
At Most Once	#18	$\sum_{0\le s<S} x_{a_i,s} \le 1$
Inelastic Actions	#19	$\sum_{0\le s<S} x_{a_{ne},s} = 1$

Figure 7: ILP Summary

4.2 Optimizing Resource Constraints

After unrolling loops, the compiler has a loop-free program it can use to generate an integer linear program (ILP) to optimize. Figure 7 summarizes the ILP variables and constraints. Below, we use the notation #k to refer to the ILP constraint or variable labeled k in Figure 7.

Action Variables. To control placement of actions, the compiler generates a set of ILP variables named $x_{a_i,s}$ (#1). The variable $x_{a_i,s}$ is 1 when the action a_i appears in stage s of the pipeline and is 0 otherwise. For instance, in the count-min sketch, there are two actions (`incr` and `min`). If we unroll a loop containing those actions twice and there are three stages in the pipeline, we generate the following action variable set.

$$\{x_{a_i,s} \quad | \quad a \in \{\texttt{incr},\texttt{min}\}, \quad 1 \le i \le 2, \quad 0 \le s < S\}$$

Register Variables. In a PISA architecture, any register accessed by an action must be placed within the same stage. Thus placement (and size) of register arrays interact with placement of actions. For each register array r and pipeline stage s, the ILP variable $m_{r,s}$ contains the amount of memory used to represent r in stage s (#2). This value will be zero in

any stage that does not contain r and its associated actions. For instance, to allocate the `cms` registers, the compiler uses:

$$\{m_{\texttt{cms}_i,s} \mid 1 \leq i \leq 2, 0 \leq s < S\}$$

Match-Action Table Variables. These variables represent the resources used by match-action tables. Similar to register variables, the variable $tm_{t_i,s}$ represents the amount of TCAM used by table t_i in stage s (#3). Note that in our current ILP, we assume that all tables, ones with and without ternary matches, use TCAM. We plan to extend the ILP so that it can choose to implement tables without ternary matches in SRAM.

Metadata Variables. The amount of metadata needed is also governed by symbolic values. If U_v is the upper bound on the symbolic value that governs the size of a metadata array, then the compiler generates a set of metadata variables d_i for $1 \leq i \leq U_v$ (#4). Each such variable will have value 1 in the ILP solution if that chunk of metadata is required and constraints described later will bound the total metadata to ensure it does not exceed the target size limits. In our running example, the bound U_v corresponds to the number of iterations of the loop that finds the global minimum value in the CMS.

Dependency Constraints. If a set of actions use the same register, they must be placed on the same stage. To do so, the compiler adds a *same-stage constraint* (#5). Similarly, if an action has a data or control dependency on another action, the two must be placed in separate stages. If there is an exclusion edge between actions a_i and b_i, the compiler creates a constraint to prevent these actions from being placed in the same stage (#6). If there is a precedence edge between actions a_i and b_i, the compiler creates a constraint forcing a_i to be placed in a stage before b_i (#7).

Conditional Constraints. In some cases, as it happens in our CMS example, multiple loops are governed by the same symbolic values. Hence, iterations of one loop (and the corresponding actions/metadata) exist if and only if the corresponding iterations of the other loop exist. Moreover, if any action within a loop iteration cannot fit in the data plane, then the entire loop iteration should not be instantiated at all. Conditional constraints (#8) enforce these invariants.

Resource Constraints. We generate ILP constraints for each of the resources listed in §2.1. Our ILP constraints reflect the memory limit per stage (#9) and the fact that memory and corresponding actions must be co-located (#10). The compiler also generates constraints to ensure that each register array in an array of register arrays has the same size (#11). Moreover, the ILP includes a constraint to guarantee that the TCAM tables in a stage fit within a stage's resources (#12).

To enforce limits on the number of stateful and stateless ALUs used in each stage, we assume that the target provides two functions $H_f(a_i)$ and $H_l(a_i)$ as part of the target specification. These functions specify the number of stateful and stateless ALUs, respectively, required to implement a given action a_i on the target. Given that information, the compiler generates constraints to ensure that the total number of ALUs used by actions in the same stage do not exceed the available ALUs in a stage (#13, #14).

To track the use of PHV, constraint #15 ensures d_i is 1 whenever the action a_i (which accesses data d_i) is used in loop iteration i. To limit the total number of PHV bits, constraint #16 sums the size in bits ($bits_d$) of the metadata d associated with iteration i and enforces it to be within the PHV bits available to elastic program components ($P - P_{fixed}$, where P_{fixed} is the amount of metadata not present in elastic arrays). Finally, each stage in the PISA pipeline can perform a limited number of hash functions. To capture that, the compiler generates constraint #17, which ensures that the number of actions including a hash function h in each stage does not exceed the available number of available hashing units N.

Other Constraints. The compiler generates a constraint so that each action a_i is placed at most once (#18). Moreover, the compiler ensures that each *inelastic* action a_{ne} (*i.e.,* an action not encapsulated in a loop bounded by a symbolic value) must be placed in the pipeline (#19). Finally, any assume statements appearing in the P4All program are included in the ILP.

4.3 Limitations

Our current ILP formulation assumes each register array and match-action table can be placed in at most one stage. However, a PISA target could conceivably spread a single array or table across multiple pipeline stages. To accommodate multi-stage arrays or tables, we can relax the ILP constraint on placing actions in at most one stage (#18).

Moreover, some compilers further optimize the use of the PHV. For example, after a metadata field has been accessed, the PHV segment storing that field could be overwritten in later stages if the metadata were never accessed again. Our prototype does not yet capture PHV field reuse.

P4All optimizes with mostly static criteria. We do not consider any dynamic components, unless a programmer incorporates a workload-dependent parameter in their objective function. P4All also does not support elastic-width fields or parameterized packet recirculation. We leave these features, as well as PHV reuse, for future work.

5 Prototype P4All Compiler

In this section, we describe our prototype P4All compiler, written in Python.

Target specification. We created a target specification for the Intel Tofino switch, based on product documentation. The specification captures the parameters in Section 2.1 and the H_f and H_l functions that specify the number of ALUs required to implement a given action. Since the Tofino design is proprietary, our specification unquestionably omits some low-level constraints not described in the documentation; with knowledge of such constraints, we could augment our target specification and optimization framework to handle them.

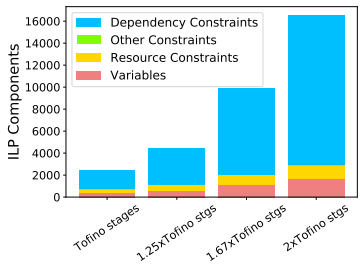
Applications	P4All Code	Compile Time (sec)	ILP (Var, Constr)
Linear Objective			
IPv4 Forwarding + Stateful Firewall	217	0.4	(192, 1026)
BeauCoup	541	0.1	(672,7511)
Precision	166	25.7	(1316, 18969)
NetChain	242	27.9	(252, 3278)
Elastic Switch.p4	804	0.2	(1080, 21581)
Non-Linear Objective			
Key-value store (KVS)	127	15.4	(168, 857)
Count-min sketch (CMS)	82	1.8	(396, 1994)
KVS + CMS (Section §3)	170	27.9	(586, 2815)
Non-Elastic Switch.p4 + CMS	853	17.5	(1498, 23575)
SketchLearn	445	2.4	(768, 880)
ConQuest	362	5.8	(612, 3734)
Multivariate Objective			
Bloom filter	70	513.6 (longest) 170.0 (avg)	(240, 308) (132, 191)
CMS + Bloom	223	67.3 (longest) 38.1 (avg)	(658, 2266) (550, 2149)

Figure 8: P4All applications, showing the lines of code in the P4All implementation. For structures with multiple instances, the last two columns give statistics for the single instance with the *longest* compile time and the *average* of all instances.

Compute upper bounds for symbolic values. To compute upper bounds and unroll loops, our prototype must analyze P4 dependencies. To facilitate this, we use the Lark toolkit [1] for parsing. We have also written a Python program that finds dependencies between actions and tables and outputs the information in a format our ILP can ingest. At the moment, we only produce precedence edges. As a result, we do not process exclusion edges, treating all edges as precedence edges. We plan to upgrade this in the future.

Generate and solve ILP. Our prototype generates the ILP with variables and constraints in Figure 7, as well as the objective function. We then invoke the Gurobi Optimizer [18] to compute a concrete assignment for each symbolic value. We then use these values to generate the unrolled P4 code.

P4 compiler. After the compiler converts the P4All program into a P4 program, we invoke the (black box) Tofino compiler to compile the P4 program for execution on the underlying Tofino switch. If our experiments initially fail to compile to the Tofino switch because of proprietary constraints, we adjust our target specification and added `assume` statements to further constrain the memory allocated to register arrays. Ideally, the P4All compiler would be embedded within a target-specific compiler to automatically incorporate the proprietary constraints, without our needing to infer them.



(a) Number of ILP variables and constraints for CMS as stages increase.

Num Stages	ILP Time (s)
Tofino	1.8
1.25xTofino	4.5
1.67xTofino	53.1
2xTofino	216.0

(b) ILP completion time for CMS as stages increase.

Figure 9: ILP performance as number of available stages increases.

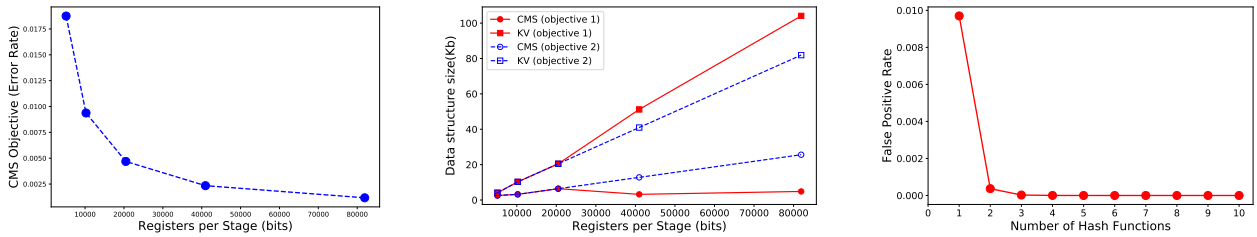
6 Performance Evaluation

6.1 Compiler Performance

Figure 8 reports the sizes of the constraint systems, and the compile times, for benchmark applications when compiled against our Tofino resource specification. We choose applications with a variety of features, including elastic TCAM tables (Switch.p4), multivariate objectives (Bloom filter), elastic and non-elastic components (IPv4 forwarding and stateful firewall), and multiple elastic components (KVS and CMS, CMS and Bloom filter). In our experiments, we found that the choice of objective function greatly impacts performance. For example, a non-convex objective function results in a mixed integer program (MIP) instead of an ILP, which significantly increases solving time. On the other hand, our applications with linear objective functions (e.g., Switch.p4, BeauCoup) typically had smaller compile times. Additionally, increasing the step size for an objective (i.e., reducing the number of values provided to the ILP) decreases compile time.

For the data structures we evaluated with objective functions with multiple variables (*e.g.*, Bloom filter), our compiler created multiple instances of the optimization problem. We report the average compile time and the average number of ILP variables and constraints *for each instance*, along with the statistics for the largest instance. Our prototype compiler is not parallelized, but could easily be in the future, allowing us to solve many (possibly all) instances at the same time. Compile times of each ILP instance for the Bloom filter application range from roughly one second to 8.5 minutes.

Compile time increases as we increase the number of elastic elements in a P4All program. We evaluate ILP performance by observing the solving time as we increase the number of elastic elements in a program. Compilation for a single elastic sketch completed in about 10 seconds, while compilation for



(a) CMS error rate as memory increases. **(b)** CMS and KVS sizes for different objectives. **(c)** Bloom filter false positive vs. # hash functions.

Figure 10: Elasticity of P4All

four sketches took over 30 minutes.

The number of constraints also affects compile time. The Bloom filter had the fewest ILP constraints, as it had no dependent components, and it alone had the largest compilation time. The reason for this is that the smaller number of constraints may lead to a more difficult optimization problem.

When we increase the available resources on the target, we generate a larger optimization problem, with more variables and constraints. Figure 9a the change in the number of constraints and variables as we increase the number of available stages on the target. Most of the resource and other constraints (*e.g.*, TCAM size, hash units, at most once, etc.) are linearly proportional to the stages. The dependency constraints are the only constraints that do not increase linearly with the stages. For a single P4All action, we create an ILP variable for each stage. However, the variables for CMS are not linearly proportional to the stages because as we increase stages, the upper bound on the actions also increases, resulting in more variables. Similarly, the ILP completion time increases super linearly with the number of stages (Figure 9b).

Some applications may have both elastic and non-elastic components. In our evaluations, we found that this did not significantly impact compile time. When we combined an elastic CMS and Switch.p4 (with fixed-size TCAM tables), the compile time was 17 seconds. Our compiler requires that all non-elastic portions of the program get placed on the switch, or the program will fail to compile.

Hand-written vs P4All-generated P4 To investigate whether P4All-generated P4 was competitive with hand-written P4, we examined a few P4 programs written by hand by other programmers and compared those programs with the P4 code generated from P4All. When we compare the number of registers used by the manually-written BeauCoup and the P4All-generated BeauCoup, we find they are exactly the same. ConQuest is made up of sketches, so we use the same objective function described in §3. With that function, our compiler tries to allocate as many registers as possible, and allocates all available space to sketches, as more registers means lower error. Examining the ConQuest paper in more depth, however, shows that the accuracy gains are minimal after a certain point (2048 columns). To account for this, we easily adjust the objective function, and as a result, the compiled code uses exactly 2048 columns as in the original. This experiment illustrates the power of P4All beautifully. On one hand, our first optimization function is highly effective—it uses up all available resources. On the other hand, when new information arrives, like the fact that empirically, there are diminishing returns beyond a certain point, we need only adjust the objective function to reflect our new understanding of the utility. None of the implementation details need change. While this analysis is admittedly ad hoc, our findings here suggest that P4All does not put programmers at a disadvantage when it comes to producing resource-efficient P4.

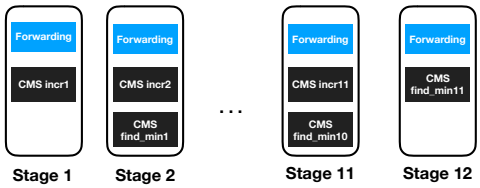
6.2 Elasticity

In this section we measure how utility of data structures vary as resources are made available. Figure 10a shows how the error rate of a CMS decreases as we increase the available registers in each stage. Figure 10b shows how the sizes of a KVS and CMS change for different objective functions. We use the objective functions for KVS hit rate and CMS error rate as described in Figure 5. The first objective function $0.8 * (kv\_obj) - 0.2 * (cms\_obj)$ gives a higher weight to the KVS hit rate, while the second $0.2 * (kv\_obj) - 0.8 * (cms\_obj)$ gives a higher weight to the CMS error rate.

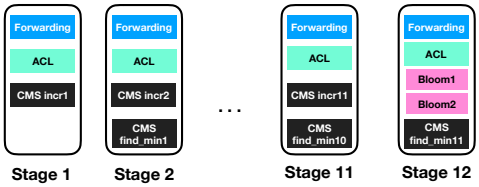
For multi-variate functions, the compiler generates multiple instances of the optimization problem, and chooses the solution to the instance with the best objective. In Figure 10c, we show the objective (false positive rate) from the instances of optimization for a Bloom filter. In each instance, the compiler increases the number of hashes used. The objective decreases for each instance, but not by much after the first instance.

6.3 Case Study

In a conversation with a major cloud provider, the researchers expressed interest in hosting a multiple applications on the same network device, which must include forwarding logic. We designed P4All for exactly such scenarios—elastic structures allow new applications to fit onto a shared device. We consider a simple case study oriented around this problem.



(a) Switch layout with forwarding tables and a CMS.



(b) Switch layout with forwarding tables, ACL tables, CMS, and Bloom filter used in stateful firewall.

Figure 11: Switch program layouts.

To do so, we started with the IPv4 forwarding code from switch.p4, but the size of the table is defined symbolically in P4All. We then added a CMS for heavy hitter detection. Figure 11a illustrates the layout: The forwarding tables utilize all of the TCAM resources, and the CMS uses registers.

Next, to demonstrate the flexibility and modularity of our framework, we add access control lists (ACLs), which use match-action tables, and squeeze in a stateful firewall, using Bloom filters, similar to the P4 tutorials [2]. Using P4, the programmer would manually resize the CMS and forwarding tables so the new applications could fit on the switch, but by using P4All, we do not have to change our existing code at all. To write ACLs with elastic TCAM tables, we modify the code in switch.p4 to include symbolic table sizes. Our compiler automatically resizes the elastic structures to fit on the switch, resulting in the layout in Figure 11b. The forwarding tables and ACLs now share the match-action table resources, and the registers in the Bloom filter fit alongside the CMS.

7 Related Work

Languages for network programming. There has been a large body of work on programming languages for software defined networks [3, 14, 37, 44] targeted towards OpenFlow [33], a predecessor to P4 [7, 36]. OpenFlow only allows for a fixed set of actions and not control over registers in the data plane, and so these abstractions are not sufficient for P4. While P4 makes it possible to create applications over a variety of hardware targets, it does not make it easy. Domino [40] and Chipmunk [16] use a high-level C-like language to aid in programming switches. P4All also aims to simplify this process, but we enhance P4 with elastic data structures. Domino and Chipmunk optimize the data-plane layout for static, fixed-sized data structures, and P4All optimizes the data structure itself to make the most effective use of resources.

Using synthesis for compiling to PISA. The Domino compiler extracts "codelets", groups of statements that must execute in the same stage. It then uses SKETCH [42] program synthesis to map a codelet to ALUs (atoms in the paper's terminology) in each stage. If any codelet violates target constraints, the program is rejected. To improve Domino, Chipmunk [16] uses syntax-guided synthesis to perform an exhaustive search of all mappings of the program to the target. Thus, it can find mappings that are sometimes missed by Domino. Lyra [15], extends this notion to a one-big-pipeline abstraction, allowing the composition of multiple algorithms to be placed across several heterogeneous ASICs. Nevertheless, Domino, Chipmunk and Lyra map programs with fixed-size data structures, while P4All enables elastic data structures.

Compiling to RMT. Jose et al. [28] use ILPs and greedy algorithms to compile programs for RMT [8] and FlexPipe [35] architectures. These ILPs are part of an all-or-nothing compiler which attempts to place actions on a switch based on the dependencies and the sizes of match-action tables. In contrast, the P4All compiler allows for elastic structures, which can stretch or compress according to a target's available resources.

Programmable Optimization. P^{2}GO [45] uses profile-guided optimization (*i.e.,* a sample traffic trace, not a static objective function) to reduce the resources required in a P4 program. P^{2}GO can effectively prune components that are not used in a given environment; however, if unexpected traffic turns up later, P^{2}GO may have pruned needed functionality!

8 Conclusion

In this paper, we introduce the concept of *elastic data structures* that can expand to use the resources on a hardware target. Elastic switch programs are more modular than their inelastic counterparts, as elastic pieces can adjust depending on the resource needs of other components on the switch. They also are portable, as they can be recompiled for different targets.

P4All is a backwards-compatible extension of P4 that includes symbolic values, arrays, loops and objective functions. We have developed P4All code for a number of reusable modules and several applications from the recent literature. We also implement and evaluate a compiler for P4All, demonstrating that compile times are reasonable and that auto-generated programs make efficient use of switch resources. We believe that P4All and our reusable modules will make it easier to implement and deploy a range of future data-plane applications.

Acknowledgments

We thank the anonymous NSDI reviewers and our shepherd Costin Raiciu for their valuable feedback. This work is supported by DARPA under Dispersed Computing HR0011-17-C-0047, NSF under FMiTF-1837030 and CNS-1703493 and the Israel Science Foundation under grant No. 980/21.

References

[1] Lark parser. https://github.com/lark-parser/lark.

[2] Stateful firewall in P4. https://github.com/p4lang/tutorials/tree/master/exercises/firewall.

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 113–126. ACM, 2014.

[4] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Shir Landau Feibish, Danny Raz, and Minlan Yu. Routing oblivious measurement analytics. In *IFIP Networking*, pages 449–457, 2020.

[5] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. PINT: Probabilistic in-band network telemetry. In *ACM SIGCOMM*, pages 662–680, 2020.

[6] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *IEEE International Conference on Network Protocols*, pages 313–323, Sep. 2018.

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

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

[9] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *IEEE INFOCOM*, pages 126–134, 1999.

[10] Xiaoqi Chen, Shir Landau Feibish, Mark Braverman, and Jennifer Rexford. BeauCoup: Answering many network traffic queries, one memory update at a time. In *ACM SIGCOMM*, pages 226–239, 2020.

[11] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A. Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *ACM SIGCOMM Conference on Emerging Networking EXperiments and Technologies*, pages 15–29. ACM, 2019.

[12] Saar Cohen and Yossi Matias. Spectral bloom filters. In *ACM SIGMOD*, pages 241–252. ACM, 2003.

[13] Graham Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In Hillol Kargupta, Jaideep Srivastava, Chandrika Kamath, and Arnold Goodman, editors, *SIAM International Conference on Data Mining*, pages 44–55. SIAM, 2005.

[14] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ACM SIGPLAN International Conference on Functional Programming*, pages 279–291. ACM, 2011.

[15] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous ASICs. In *ACM SIGCOMM*, pages 435–450, 2020.

[16] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *ACM SIGCOMM*, page 44–61, 2020.

[17] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*, pages 357–371. ACM, 2018.

[18] Gurobi Optimization. Gurobi optimizer reference manual. http://www.gurobi.com, 2019.

[19] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *ACM Symposium on SDN Research*, 2018.

[20] Rob Harrison, Shir Landau Feibish, Arpit Gupta, Ross Teixeira, S. Muthukrishnan, and Jennifer Rexford. Carpe elephants: Seize the global heavy hitters. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure*, pages 15–21, 2020.

[21] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. Elastic switch programming with P4All. In *ACM Workshop on Hot Topics in Networks*, page 168–174, 2020.

[22] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 161–176, Boston, MA, February 2019.

[23] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. SketchVisor: Robust network measurement for software packet processing. In *ACM SIGCOMM*, pages 113–126, 2017.

[24] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *ACM SIGCOMM*, pages 576–590, 2018.

[25] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Life in the fast lane: A line-rate linear road. In *ACM Symposium on SDN Research*, 2018.

[26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-RTT coordination. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 35–49, Renton, WA, April 2018.

[27] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soule, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Symposium on Operating System Principles*, 2017.

[28] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *USENIX Conference on Networked Systems Design and Implementation*, pages 103–115, 2015.

[29] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 312–327. Springer Berlin Heidelberg, 2010.

[30] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 311–324, Santa Clara, CA, March 2016.

[31] Zaoxing Liu, Ran Ben Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *ACM SIGCOMM*, pages 334–350, 2019.

[32] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM*, pages 101–114, 2016.

[33] Nick McKeown, Thomas E. Anderson, Hari Balakrishnan, Guru M. Parulkar, Larry L. Peterson, Jennifer Rexford, Scott Shenker, and Jonathan S. Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[34] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *ACM SIGCOMM*, pages 15–28, 2017.

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

[36] P4 Language Consortium. $P4_{16}$ language specifications, 2018. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf.

[37] Cole Schlesinger, Michael Greenberg, and David Walker. Concurrent NetCore: From policies to pipelines. In *ACM SIGPLAN International Conference on Functional programming*, pages 11–24, 2014.

[38] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *USENIX Networked Systems Design and Implementation*, pages 67–82, March 2017.

[39] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 1–16, Renton, WA, April 2018.

[40] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM*, pages 15–28, 2016.

[41] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *ACM SIGCOMM Symposium on SDN Research*, pages 164–176, 2017.

[42] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Architectural Support for*

Programming Languages and Operating Systems, pages 404–415, 2006.

[43] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 530–541, 2014.

[44] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *ACM SIGCOMM*, volume 43, pages 87–98, August 2013.

[45] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. P2GO: P4 profile-guided optimizations. In *ACM Workshop on Hot Topics in Networks*, page 146–152, 2020.

[46] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *ACM SIGCOMM*, pages 561–575, 2018.

[47] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, centralized lock management using programmable switches. In *ACM SIGCOMM*, pages 126–138, 2020.

[48] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow event telemetry on programmable data plane. In *ACM SIGCOMM*, pages 76–89, 2020.