Elastic Switch Programming with P4All

Mary Hogan, Shir Landau-Feibish, Mina Arashloo, Jennifer Rexford, David Walker, Robert Harrison

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 present P4All, an extension of P4 that allows programmers to define elastic data structures that stretch automatically to make use of available switch resources. 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. We have implemented a P4All compiler that identifies the program dependencies and uses them, along with the target resource constraints, to generate optimized P4 code. Our evaluation shows that compile times are small and that P4All makes effective use of limited switch resources.

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 [4, 29], enable new ideas to come to fruition quickly.

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 real applications need to combine traffic monitoring, key-value stores, packet forwarding, and more. Yet, deciding how to divide precious switch resources across multiple kinds of functionality is 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. That is too bad because, as shown in Figure 1, many applications use similar structures. Despite the great potential for reuse, these structures are rarely shared between developers as libraries. 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.

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 [18] that caches popular keys by combining (i) a count-min sketch [7] (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 compiler should maximize. For example, 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 processing 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) 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 construct operations over
<table>
<thead>
<tr>
<th>Module</th>
<th>Used in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-value store/</td>
<td>Precision [2], Sonata [10], Network-Wide HH [12], Sketchvisor [14],</td>
</tr>
<tr>
<td>hash table</td>
<td>Linear-Road [16], NetChain [17], NetCache [18], FlowRadar [21], HashPipe [34], Elastic</td>
</tr>
<tr>
<td></td>
<td>Sketch [39]</td>
</tr>
<tr>
<td>Hash-based matrix</td>
<td>Sketchvisor [14], NetCache [18], Ni-trosketch [23], UnivMon [24],</td>
</tr>
<tr>
<td></td>
<td>Sharma et al. [31], Fair Queueing [32], Elastic Sketch [39]</td>
</tr>
<tr>
<td>Hierarchical sketch</td>
<td>Sketchlearn [15]</td>
</tr>
<tr>
<td>Bloom filter</td>
<td>NetCache [18], FlowRadar [21], SilkRoad [26], Sharma et al. [31]</td>
</tr>
<tr>
<td>ID indexed table</td>
<td>Blink [13]</td>
</tr>
</tbody>
</table>

Figure 1: PISA data structures

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 demonstrate the feasibility of our design, we have implemented a compiler for P4All. The compiler operates in two main stages. First, it computes an upper bound on the number of possible iterations of the loops, so it can produce a simpler optimization problem over unrolled, loop-free code. This upper bound is computed by simultaneously (and conservatively) analyzing the dependency structure of the loop bodies and their resource utilization. Once the upper bounds have been computed, a second phase of the compiler can unroll the loops to those bounds and generate an integer linear program (ILP) that accurately optimizes the resource utilization of the loop-free code for a particular target.

In principle, a compiler like ours could generate the low-level configuration of the PISA hardware target. After all, to compute assignments for the symbolic values, our P4All compiler already generates a mapping of program elements to stages in the data plane. In practice, today’s PISA hardware platforms are proprietary, so our P4All compiler cannot generate the low-level configuration directly. In our prototype, our compiler takes a target specification (that summarizes the target’s capabilities and resources) as input and uses it to generate a P4 program that a target-specific P4 compiler can, in turn, map to the switch hardware. Given the complexity of programmable switch hardware, our target specification inevitably omits some target-specific constraints; with more intimate knowledge of the target hardware, developers could extend our ILP to account for these additional constraints.

We demonstrate the utility of our language and system by (1) developing a library of reusable elastic modules (e.g., Bloom filter, count-min sketch, and key-value store) and (2) building a range of elastic applications from the literature by combining these modules. Our experiments show that the P4All compiler runs quickly and produces concrete P4 programs that are competitive with hand-optimized code. In summary, our contributions are:

- The design of the P4All language, a backward-compatible extension to P4 that enables elastic network programming,
- the design and implementation of an optimizing compiler from P4All to P4,
- the creation of reusable modules for common data structures, and examples of combining these modules to create more sophisticated applications, and
- an evaluation of our system for a range of applications, compiled to the Barefoot Tofino switch, that shows that our P4All system is expressive and efficient.

We begin by discussing data-plane resources (§ 2). We then describe the difficulty of programming for PISA and the constructs P4All provides to remedy these problems (§ 3). Next, we describe the P4All compiler (§ 4) and our prototype (§ 5), followed by their evaluation (§ 6). The paper ends with an overview of related work (§ 7) and a conclusion section (§ 8) that discusses future research directions.

Ethics: This work does not raise any ethical issues.

2 PISA DATA-PLANE RESOURCES

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

Between parser and deparser sits the packet-processing pipeline, which is composed of a series of stages. Each stage contains a fixed set of resources, which bounds the amount of computation it may perform on a packet before the packet moves on to the next stage. Actions define what computation should happen on the packets going through the pipeline. To trigger the execution of an action on all or a subset of packets, each stage contains tables of match-action rules. A rule can fire on all packets, or only when a specific bit pattern matches data in the PHV. The rule’s actions are performed by the ALUs associated with a stage. In general, a stage has a fixed number of ALUs, bounding the computation a stage may perform. In addition, all actions executed by a single stage happen concurrently.

To perform two actions in sequence (because execution of one action depends on the result of the other), one must use two different stages. Finally, each stage is typically associated with a finite set of registers. These registers serve as persistent memory, recording and preserving information, such as counts, across packets. Because packets are processed in a feed-forward fashion, they can only access memory in the current stage and cannot access memory from previous stages. Actions that get or set registers are called stateful actions. Only a small subset of the ALUs in each stage (typically less than five) support stateful actions. The rest of the ALUs handle actions that get or set PHV values, and are referred to as stateless actions. Figure 3 summarizes the parameters that define a generic PISA model.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Number of pipeline stages</td>
</tr>
<tr>
<td>M</td>
<td>Available register memory per stage (bits)</td>
</tr>
<tr>
<td>F</td>
<td>Number of stateful ALUs per stage</td>
</tr>
<tr>
<td>L</td>
<td>Number of stateless ALUs per stage</td>
</tr>
<tr>
<td>P</td>
<td>Size of packet header vector (PHV) (bits)</td>
</tr>
</tbody>
</table>

Figure 3: PISA parameters

3 PISA PROGRAMMING PERILS

Programming PISA devices is difficult because the resources available in each stage 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.

The P4 language attempts to alleviate a number of these problems by providing a layer of abstraction above PISA. A P4 compiler maps these higher-level abstractions down to the PISA architecture and organizing 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 other words, they must redo many of the jobs of the compiler themselves. 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. 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.

3.1 Tackling 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 [18], 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 focuses on implementing the CMS, a probabilistic data structure that uses multiple hash functions to keep approximate frequencies for a stream of items in sublinear 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 approximation may overestimate the number of occurrences of a packet \( x \) if there are hash collisions. Hence, 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 tackle the question of how many resources to assign to the CMS, including how much memory to allocate and how to split up that memory into separate rows. Resource allocation becomes even harder when the programmer has to grapple with dividing resources between multiple structures. In Figure 4 we see the quality of NetCache for different resource combinations of the key-value store and the CMS. Achieving the highest quality requires determining the cost of implementing the hash/increment/store/min-value computations and whether some operations can run in parallel or whether they require sequential pipeline stages. Moreover, the module implemented by the programmer may not be the only piece that consumes resources in the programmable switches; essential functions, such as forwarding and load balancing, also consume resources. Even if the engineer is an expert and familiar with the particular target device, estimating the resources needed to integrate all of these modules is challenging, given all the constraints.

 Unsure how to best allocate switch resources for the CMS, our engineer simply guesses that using four hash functions \((r = 4)\) and 2048 columns would likely fit within the constraints of their target hardware. To determine if this allocation actually works on the target, the engineer must write the full program and compile it. If it fits, then great; if not, then the engineer enters a tedious “trial and error” process of rewriting and recompiling the program.

Figure 5 presents a fragment of the P4 program developed by our engineer to solve the problem. Lines 1-7 declare the metadata needed to look up the estimate in the CMS for the count of 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 can each store values represented by 32 bits. Lines 15-21 and 23-27 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 29-46), used to dictate which actions and/or rule tables apply to each packet, the program first executes all the hash actions, computing and storing counts for each hash function, and then compares those counts to each other looking for the minimal one.

Upon reviewing the code for this component of the NetCache application, some of the deficiencies of P4 should immediately be apparent. First, there is a great deal of repeated code: Repeated data structure definitions, repeated action definitions, and repeated 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 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 when 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 the programmer had to choose magic constants (like 2048) and test themselves whether such constants lead to programs that can compile or not.

### 3.2 Tackling NetCache in P4All

P4All improves upon P4 by making it possible to construct and manipulate 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.

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 the use of this methodology by examining the definition of an elastic count-min sketch and its use in an elastic NetCache application.

Figure 5: Count-Min Sketch in P4 (16)

Figure 6: NetCache and Count-Min Sketch in P4All

3.2.1 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 numbers of the columns (i.e., the range of the hash). Such parameters are defined as symbolic values:

```
symbolic int rows;
symbolic int 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 [20], Sketch [35], or Rosette [37], 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 as follows.

\begin{verbatim}
assume 0 <= rows && rows < 4
\end{verbatim}

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.

### 3.2.2 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 for the user.

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:

\begin{verbatim}
register<bit<32>>[cols][rows] cms;
\end{verbatim}

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 the information from a single register in a row. Within the metadata struct, we define:

\begin{verbatim}
bit<32>[rows] index;
bit<32>[rows] count;
\end{verbatim}

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

### 3.2.3 Define Elastic Operations

Because elastic data structures can stretch or contract to fit available resources, elastic operations over these 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 putting the result in 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.

\begin{verbatim}
/* 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];
            }
        }
    }
}
\end{verbatim}

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.

\begin{verbatim}
control NetCache( ... ) {
    apply {
        hash_inc.apply(...);
        find_min.apply(...);
        ... }
}
\end{verbatim}

### 3.2.4 Manage Competing Resource Needs

Data structures written for programmable switches are valid at a range of shapes and sizes. In the CMS example above, there may be multiple assignments to rows and cols which could fit within the resource constraints of the switch. Finding the right parameters becomes even more difficult when multiple data structures are involved. In the case of NetCache, after defining a count-min sketch, the programmer still needs to define and optimize a key-value store, for instance.

To automate this process, P4All allows programmers to define a utility function that expresses the relationship between the utility of a data structure and its size (as defined by symbolic values). These utility functions are simply arithmetic expressions that include one or more of the symbolic values declared in the program. The P4All compiler finds instances of the symbolic values that optimize the utility function subject to the constraint that the resulting program can fit within switch resources.

For instance, to optimize the count-min sketch, the programmer defines a function that maximizes the number of rows or the number of columns or finds a balance between the two. Similarly, to optimize NetCache the programmer should define a function that balances between the count-min sketch and the key-value store.
For example, in NetCache, it may be more important to reserve room for the key-value store than the count-min sketch. To do so, a programmer could use the following utility function, which slightly prioritizes allocating resources to the key-value store over the CMS:

\[
\text{optimize} \quad 0.4 \cdot (\text{rows} \times \text{cols}) + 0.6 \cdot (\text{kv\_items})
\]

This utility function results in the layout in Figure 7 and the optimal configuration in Figure 4. The CMS will have two rows in the first stage, while the NetCache key-value store fills the following nine stages. We evaluate this particular configuration in §6. In a different application, the count-min sketch may need a higher proportion of resources. If so, the programmer can rewrite the utility function. Still, doing so does not affect the functional correctness of the CMS data structure and hence such rewrites do not generate logical errors, for instance. Moreover, this performance tuning occurs in one centralized place rather than being spread throughout the application, and hence, from a software engineering perspective, leads to a superior process.

**Summary.** In this section, we introduced a number of linguistic mechanisms new to P4All: symbolic values, arrays, loops and utility functions. We also developed a programming methodology that explains how to use those features in combination to allocate and operate on elastic data structures. Figure 6 consolidates the pieces introduced in the previous paragraphs, presenting (key elements of) the count-min sketch application as a whole.

### 4 COMPILING P4ALL PROGRAMS

Compiling a traditional P4 program requires the compiler to assign program elements to stages in the target’s pipeline, while observing dependencies and resource constraints [19]. P4All’s use of symbolic values makes the compilation process more challenging. The mapping of program elements to stages now depends on the choice of the symbolic values, and yet the best assignment of symbolic values depends on the available resources. We resolve this apparent circularity by (i) computing upper bounds for the symbolic values based on a dependency analysis and (ii) constructing an integer linear program that determines the concrete values and the mapping of the resulting P4 program elements to stages.

#### 4.1 P4All Compiler Architecture

The P4All compiler takes, as input, a P4All program and a specification of the target’s resources (i.e., the parameters from Figure 3) and capabilities of the ALUs, as shown in Figure 8. The compiler outputs a P4 program (with a concrete assignment for each symbolic value), as well as a mapping of P4 program elements to stages in the target’s pipeline. The P4All compiler could be a module in a target-specific compiler that uses the mapping information to generate the final, low-level configuration of the target. Alternatively, the concrete P4 program can be an input to a (black box) target-specific P4 compiler; we take this approach in our P4All prototype for the Barefoot Tofino target (§5).

The P4All compiler first analyzes the control and data dependencies between actions in the program, as well as the target resources, to compute an upper bound on the number of times each loop must be unrolled (§4.2). 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 (§4.3). The upper bound ensures
we use the CMS program in Figure 6 as a running example. The picture above shows the body of a for loop unrolled three times. This picture contains a simple path of length four (incr_1, min_1, incr_3, min_3) which would not fit in the three-stage pipeline. The purple path with length 3, which covers just the first two loop iterations, will fit. Hence, the loop is unrolled twice before a finer analysis via ILP.

Figure 9: Loop unrolling. Assume the target has three stages. The picture above shows the body of a for loop unrolled three times. This picture contains a simple path of length four (incr_1, min_1, min_2, min_3) which would not fit in the three-stage pipeline. The purple path with length 3, which covers just the first two loop iterations, will fit. Hence, the loop is unrolled twice before a finer analysis via ILP.

this integer linear program is “large enough” to consider all possible placements of data and to 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 utility function when other constraints are accounted for. The resulting ILP solution is a layout of the data-plane algorithm, 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 6 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).

4.2 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 a 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 \( v \).

The nodes (\( n \)) of the dependency graph \( G_v \) are annotated with the set of actions \( A_n \) that access the same register and thus must be placed in the same stage.

There are two types of edges in our dependency graphs: (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 \( n_1 \rightarrow n_2 \)) 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 \( n_1 \rightarrow n_2 \) 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 (\( n_1 \leftarrow n_2 \)) 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 \( a_2 \) 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 9 shows the dependency graph for rows from our running CMS example. Only the incr_1 actions access register arrays, and they all access different register arrays. Thus, each node represents only one action. There is a precedence edge from incr_1 to min_1 as the former writes to the same metadata variable read by the latter. Thus, incr_1 must be placed in a stage preceding min_1. There are exclusion edges between each pair of min_1 and min_2 because they are commutative but write to the same metadata fields: min_1 sets the metadata variable tracking the global minimum meta.min to the minimum of its current value and the \( i \)th row of the CMS (meta.count[1]).

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 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
After unrolling loops, the compiler has a loop-free program. Assume also the valid range of values for both variables. 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 containing those actions twice and there are three stages. The compiler generates a set of ILP variables named \( k \) to refer to the ILP constraint values. Figure 10 summarizes the ILP variables and constraints.

### 4.3 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 10 summarizes the ILP variables and constraints. Below, we use the notation \(#k\) to refer to the ILP constraint or variable labeled \( k \) in Figure 10.

#### Action Variables.

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

\[
\{ x_{a_i, s} \mid a \in \{ \text{incr, min} \}, 1 \leq i \leq 2, 0 \leq s < S \}
\]

#### Memory 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 \( s \) (this value will be zero in any stage that does not contain \( r \) and its associated actions). For instance, to allocate the \( \text{cms} \) registers, the compiler uses:

\[
\{ m_{\text{cms}, i} \mid 1 \leq i \leq 2, 0 \leq s < S \}
\]

#### 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 \( \{ d_i \mid 1 \leq i \leq U_v \} \). Each such variable will have value 1 in the ILP solution if that chunk of meta data is required (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 \textit{same-stage constraint} (#4). 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 \( a_j \), the compiler creates a constraint to prevent these actions from being placed in the same stage (#5). If there is a precedence edge between actions \( a_i \) and \( a_j \), the compiler creates a constraint forcing \( a_i \) to be placed in a stage before \( a_j \) (#6).

#### Conditional Constraints.

In the CMS, multiple loops are governed by the same symbolic values. Hence, iterations of
one loop (and the corresponding actions/metadata) exist if and only if 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 (#7) enforce these invariants.

**Resource Constraints.** Each target has a limited amount of register memory per stage (M), the number of ALUs used to for stateful actions (F), the number of ALUs that handle stateless actions (L), and total number of bits in the PHV (P).

Our ILP constraints reflect the memory limit per stage (#8) and the fact that memory and corresponding actions must be co-located (#9). The compiler also generates constraints to ensure that each register array in an array of register arrays has the same size (#10).

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 this 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 (#11, #12).

To track the use of PHV, constraint #14 ensures \( d_i \) is 1 whenever the action \( a \) (which accesses data \( d \) is used in loop iteration \( i \)). To limit the total number of PHV bits, constraint #13 sums the size in bits \( \text{bits}_d \) of the metadata \( d \) associated with iteration \( i \) and enforces it to be within the PHV bits available to elastic program components \( P - P_{\text{fixed}} \), where \( P_{\text{fixed}} \) is the amount of metadata not present in elastic arrays.

**Other Constraints.** The compiler generates a constraint so that each action \( a_i \) is placed at most once (#15). It also generates constraints so that iterations are placed in order. In other words, an action \( a_j \) in the second iteration of a loop cannot be placed unless an action of the first iteration \( a_i \) has been placed (#16). Moreover, the compiler ensures that each inelastic action \( a_{\text{ne}} \) (i.e., an action not encapsulated in a loop bounded by a symbolic value) must be placed in the pipeline (#17). Finally, any assume statements appearing in the P4All program are included in the ILP.

4.4 Limitations

**Match-Action tables.** Our current ILP formulation does not consider the placement of match-action tables in the pipeline. Existing P4 compilers already solve the table-allocation problem under target constraints, and the techniques for doing so optimally are well-understood [19]. However, we believe there is no fundamental reason why we could not incorporate the additional constraints needed to for deciding match-action table placement into our current ILP formulation.

**Hash function units.** A target may impose limitations on the number and placement of hash function units, which would require adding an additional constraint to the ILP.

**Spreading register arrays.** Our current ILP formulation assumes each register array can be placed in at most one stage. However, a PISA target could conceivably spread a single register array across multiple pipeline stages. To accommodate multi-stage register arrays, we can relax the ILP constraint on placing actions in at most one stage (#9).

**PHV reuse.** Some compilers may 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 and we leave this optimization as future work.

5 PROTOTYPE P4ALL COMPILER

In this section, we briefly describe our prototype P4All compiler, written in Python, based on the design in Figure 8.

**Target specification.** We created a target specification for the Barefoot Tofino switch, based on product documentation. The specification captures the parameters in Figure 3 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.

**P4All parser.** As a first step in the compilation process, we parse the P4All program to identify symbolic values and elastically-defined data structures.

**Compute upper bounds for symbolic values.** To compute upper bounds and unroll loops, our prototype must analyze P4 dependencies. To facilitate this dependency analysis, we use a utility supplied by the Barefoot Tofino compiler. This utility generates the dependencies the compiler infers between actions and related metadata and outputs the information in a format our tool can ingest. At the moment, the utility only produces precedence edges. As a result, we do not process exclusion edges in our prototype, treating all edges as precedence edges. If supplied with better dependency information, we could upgrade the analysis.

**Generate and solve ILP.** Our prototype generates the ILP with variables and constraints in Figure 10, as well as the utility function. We then invoke the Gurobi Optimizer [11] 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.
In our experiments, our benchmark applications (Figure 11) initially failed to compile to the Tofino switch because of proprietary constraints. To remedy this, we adjusted 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.

### 6 PERFORMANCE EVALUATION

We evaluate P4All to show that the language and compiler can support a variety of realistic data-plane applications (§6.1). We then show that P4All optimally stretches programs to utilize the target’s available resources, using several different utility functions (§6.2).

#### 6.1 Benchmarking P4All Applications

To evaluate P4All, we create a library of reusable P4All modules of commonly used data structures, including count-min sketch, Bloom filter [3], key-value store, and hash table. Using these modules, we can express a variety of applications in P4All. In particular, NetCache [18] uses a count-min sketch and a key-value store, SketchLearn [15] and ConQuest [6] use multiple instances of count-min sketch, and Precision [2] uses hash tables.

Figure 11 lists the applications we have developed and provides several statistics. Each application was drawn from the literature and was originally written by the authors of the papers listed above; we ported the applications to P4All. We compare the length (in lines of code) of the original, hand-written P4 applications with the length of our corresponding P4All version.

In several cases, the P4All modules save the programmer from writing large sections of repeated code, leading to much shorter programs. This is true in NetCache and SketchLearn, for instance. In the Precision and ConQuest applications, the authors developed a system of macros to encapsulate repeated bits of code. In those applications, adding or removing actions or metadata definitions involves commenting or uncommenting parameterized macros. The end result is the difference, when measured purely in lines of code (a weak measure of code complexity), is less dramatic. Nevertheless, each line of code added or removed “does a lot” here, and repeatedly editing the program and recompiling to manually search for an optimal layout is not much less time-consuming or annoying. Either way, P4All is significantly more powerful than P4 because a single P4All program represents a family of possible P4 programs; the P4 compiler automatically selects from amongst that family of programs the one program that optimizes their utility function for a given target.

The P4All compiler is fast enough to handle the applications we have analyzed. Figure 11 reports compile times when compiled against our Tofino resource specification, which range from less than a second for the smaller benchmarks to roughly 15 seconds for the largest one. The compiler spends the majority of its time solving the integer linear program.

#### 6.2 Elasticity

As we increase the available resources on the target, the P4All compiler stretches data structures to make use of those resources effectively. As an example, Figure 12 shows how the sizes of both the key-value store and CMS in NetCache increase as we increase the amount of per-stage memory ($M$) on the target. We are still using $0.4 \cdot (rows \cdot cols) + 0.6 \cdot (kv\_items)$ as the utility function. Note as well that the key-value store items are far larger than the sketch items (the key-value items serves as a cache; the count-min items are just counts). As a result, the key-value store uses a larger proportion of the available memory. In this experiment, our target has ten stages ($S = 10$), four stateful ALUs ($F = 4$), 100 stateless ALUs ($L = 100$), and 4096 bits of PHV ($P = 4096$).

<table>
<thead>
<tr>
<th>Applications</th>
<th>P4 Code</th>
<th>P4All Code</th>
<th>Compile Time (s)</th>
<th>ILP (Var, Constr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetCache</td>
<td>741</td>
<td>286</td>
<td>13.47</td>
<td>(3876, 391077)</td>
</tr>
<tr>
<td>SketchLearn</td>
<td>366</td>
<td>88</td>
<td>0.19</td>
<td>(288, 1183)</td>
</tr>
<tr>
<td>Precision</td>
<td>283</td>
<td>266</td>
<td>0.83</td>
<td>(1488, 16097)</td>
</tr>
<tr>
<td>ConQuest</td>
<td>694</td>
<td>649</td>
<td>4.20</td>
<td>(6996, 88933)</td>
</tr>
</tbody>
</table>

Figure 11: P4All applications. The “P4 Code” and “P4All Code” columns show the lines of code of the original P4 application and the P4All implementation, respectively. The “ILP (Var, Constr)” column supplies the number of variables and constraints, respectively, in the integer linear program.

![Figure 12](image_url)

Figure 12: Sizes of NetCache data structures as resources increase. As we increase the per stage memory available on the target, the P4All compiler stretches the program to take advantage of the resources, according to a utility function.
The first utility function of NetCache to guarantee at least 8Mb of memory for algorithms. P4All also aims to simplify this programming in a C-like language to aid in programming packet-processing easy. Domino [33] and Chipmunk [9] utilize a new, high-level abstractions over a variety of hardware targets, it does not make it in these languages are not sufficient for P4.

Figure 13 illustrates how the choice of utility function affects the optimal sizes for the key-value store and CMS in NetCache on a target with 1.75Mb of memory per stage. The first utility function $0.4 \times (kv \_items) + 0.6 \times (rows \times cols)$ gives a higher weight to CMS, while the second $0.4 \times (rows \times cols) + 0.6 \times (kv \_items)$ prioritizes the key-value store. In both cases, the NetCache application stretches to an optimal size, utilizing as many resources as possible. Note that in this experiment, we have used assume statements in the P4All version of NetCache to guarantee at least 8Mb of memory for the key-value store, as recommended by the NetCache paper.

P4All relies on the programmer to provide a utility function that best represents the resource allocation preferences for different elastic components in the program. An interesting extension would involve building a system to generate utility functions automatically from expected workloads. We leave this topic to future research.

### 7 RELATED WORK

**Languages for network programming.** There has been a large body of work on developing programming languages for software-defined networks [1, 8, 30, 38]. These languages are targeted towards OpenFlow [25], a predecessor to P4 [4, 29]. OpenFlow only allows for a fixed set of primitive actions on each packet and does not allow programmatic control over register arrays in the data plane, and therefore, the abstractions in these languages are not sufficient for P4.

While P4 makes it possible to create exciting new applications over a variety of hardware targets, it does not make it easy. Domino [33] and Chipmunk [9] utilize a new, high-level C-like language to aid in programming packet-processing algorithms. P4All also aims to simplify this programming process, but we focus on enhancing P4 with elastic data structures via symbolic values and loops. Whereas Domino and Chipmunk optimize the data-plane layout for static, fixed-sized data structures, P4All optimizes the data structure itself to make the most effective use of pipeline resources.

**Using synthesis for compiling to PISA.** Previously, program synthesis has been used to simplify programming PISA-like targets. As mentioned above, Domino [33] is a programming language used to write data-plane algorithms in a C-like syntax. The Domino compiler extracts “codelets”, groups of statements that must execute in the same stage. It then uses SKETCH [35] program synthesis to map each codelet to the ALUs (atoms in the paper’s terminology) in each stage. If any of the codelets cannot be mapped to the target’s ALUs (i.e., violates target constraints), the program is rejected. To improve Domino’s compiler, Chipmunk [9] uses syntax-guided synthesis to perform an exhaustive search of all possible mappings of the entire program to the target. Thus, it can find mappings that are sometimes missed by the Domino compiler, or find better mappings. Nevertheless, both Domino and Chipmunk map programs with static, fixed-size data structures, while the P4All language and compiler enable elastic data structures via symbolic values and loops.

**Compiling to RMT.** Jose et al. [19] use both ILPs and greedy algorithms to compile packet-processing programs for the RMT [5] and FlexPipe [28] architectures. These ILPs are part of an all-or-nothing compiler which attempts to place every action in a program on a fixed-size switch based on the dependencies between actions 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.

**Verifying P4 programs:** Several tools have been developed to verify P4 programs using SMT solvers [22] and symbolic execution [27, 36]. While not currently included in P4All, we envision using similar methods to incorporate verification into our language. For instance, we hope to verify that all indices used with symbolic arrays are in bounds.

### 8 CONCLUSION

In this paper, we introduce the concept of elastic switch programs—programs that contain data structures capable of expanding to use the resources available on a particular hardware target. Elastic switch programs have a number of advantages over their inelastic counterparts. In particular, they are more modular as elastic components can stretch or contract depending on the resource needs of other components present on the switch. They also are portable—elastic software can be recompiled for a variety of different targets.

Elastic switch programs may be implemented in our new language P4All, a backwards-compatible extension of P4.
that includes symbolic values, arrays, loops and utility functions. We have experimented with P4All, building 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. In the future, we look forward to building many more libraries and applications. 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.

REFERENCES


