Incremental Specialization of Network Programs

Fabian Ruffy
Zhanghan Wang
New York University

Gianni Antichi
Politecnico di Milano and Queen Mary University of London

Aurojit Panda
Anirudh Sivaraman
New York University

Abstract

Programmable network devices process packets using limited time and space. Consequently, much effort has been spent making network programs run as efficiently as possible. One promising line of work focuses on specializing the implementation of a network program to a particular—presumed constant—control-plane configuration. However, while some parts of the control plane configurations are constant for long periods of time, others change frequently, and in bursts (e.g., due to routing table updates).

Thus, any approach that specializes a network program with respect to control-plane configurations must be *incremental*: it should be able to tell quickly whether a new control-plane update will affect the program's implementation and recompile the program only when its implementation must change. We describe several benefits of such an approach, including reducing resource use on line-rate pipelines and improving the memory footprint of packet classification. We explore our ideas with a prototype, Flay, an incremental partial evaluator that optimizes P4 programs by treating control-plane entries as constant. Flay can reduce resources in the implementations of Tofino programs. Flay can also determine in 100s of milliseconds whether a control-plane update will change a program's implementation. We conclude by outlining several avenues for future work.

CCS Concepts

• Networks → Programmable networks; **• Software and its engineering → Compilers**; **Incremental compilers**.

Keywords

Programmable Networks, SDN, Specialization, Partial Evaluation, Incremental Computation, P4, eBPF/XDP

ACM ISBN 979-8-4007-1272-2/24/11
https://doi.org/10.1145/3696348.3696870

ACM Reference Format:
Fabian Ruffy, Zhanghan Wang, Gianni Antichi, Aurojit Panda, and Anirudh Sivaraman. 2024. Incremental Specialization of Network Programs. In *The 23rd ACM Workshop on Hot Topics in Networks (HOTNETS '24), November 18–19, 2024, Irvine, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3696348.3696870

1 Introduction

Packet-processing programs on network devices (e.g., SmartNICs, switches, networking stacks) must rapidly process packets with limited resources (e.g., tables, ALUs, cores, CPU cycles), while simultaneously supporting many different features (e.g., ACLs, routing, NATs). Compilers for packet-processing languages [10, 32, 36, 46, 48, 67, 76] play an important role in determining the final resource requirements and performance of such programs. Typically, compilers translate the packet-processing program into an implementation when first authored, leaving the implementation unchanged over the program's lifetime.

This "one-and-done" approach leaves out many opportunities to improve implementation over a program's lifetime. In addition to the program's source code, the program's resource usage is also determined by control-plane configurations (e.g., ACL or forwarding rules). For instance, if an ACL table is empty, it can be removed, making room for additional features. Such *specializations* are especially beneficial for "kitchen-sink" programs that capture the union of all possible features [29, 66], where only a subset of features is active at any time. Prior projects have leveraged this observation: they treat control-plane configurations as constant inputs to a packet-processing program and introduce a specializing compiler to further optimize the implementation of the program before it is run [1, 23, 51].

In reality, however, control-plane configurations are *pseudo-constant*: many parts of the control plane only change in response to policy changes, maintenance, or failures (Fig. 1) and are thus infrequent. Other parts, however, change frequently (e.g., IP routes, NATs). Control-plane updates can also occur in bursts, with changes happening at once quickly followed by a long quiescence [37]. Given this pattern, our core claim is that any specializing compiler must be able to respecialize a program quickly when control-plane constants change. More so, because recompiling network programs is expensive and many control-plane updates do not affect

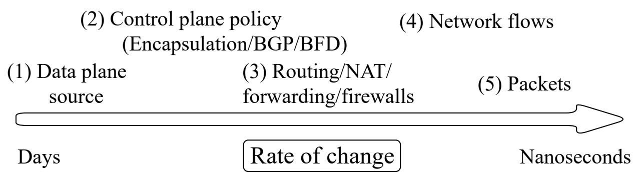


Figure 1: Varying rate of change of network program input.

the program implementation, the runtime must decide when respecialization is actually needed. Hence, to be effective, such compilers must be (1) *control-plane-triggered* so that they continuously respecialize program implementations in response to control-plane changes and (2) *incremental*, to perform as little processing as possible on program sources and control-plane configurations for each update.

We describe a design sketch for such an incremental compiler, operating as a shim layer between the network controller and the data plane (§2). We also describe several tangible benefits enabled by this approach, such as saving hardware resources, and optimizing the memory footprint and performance of packet classifiers. To demonstrate that our call for an incremental specializing compiler is feasible, we build a prototype, Flay. Flay is a partial evaluator [42] that combines several techniques (dead-code elimination, constant propagation, table inlining) to specialize P4 programs. Flay leverages the fact that P4 is a restricted domain-specific language (DSL) with a few core primitives (e.g., tables, control program) to construct SMT formulae that can quickly identify when recompilation is necessary. This allows Flay to process a control-plane update within ~100 milliseconds and avoid recompilation for all control plane updates that do not require it. By treating control variables as pseudo-constants Flay can also save pipeline resources for Tofino programs.

Flay is just a start to a broader research agenda. We outline several avenues for future work. First, for the control-plane updates that do trigger respecialization, we plan to use Flay as a vehicle to explore the tradeoff between recompilation time and specialization quality. Second, during respecialization, we are still bottlenecked by existing device compilers that monolithically compile the entire program, causing much longer compile times than necessary. We can push incremental specialization much further by (1) rearchitecting device compilers to also operate incrementally, e.g., by only recompiling the tables in the program that actually changed and (2) through hardware support for partial configuration.

2 Control-Plane-Driven Specialization

Context. A programmable networking device contains at least one programmable block (e.g., the match-action pipeline in Tofino [16] or eBPF hooks in the Linux kernel). This

Program	switch [66]	scion [22]	Beaucoup [12]	ACCTurbo [3]	DTA [45]
Time	106 s	38 s	22 s	28 s	25 s

Table 1: bf-p4c [16] compile times for Tofino P4$_{16}$ programs.

programmable block is configured by loading a binary produced by translating a network program written in a domain-specific language such as P4$_{16}$ [10], eBPF written in restricted C [30], microcode [78], or NPL [9]. Network programs written in these languages also define a control-plane interface to influence packet-processing behavior of the program during runtime. The control-plane interface is specific to each program. Prominent examples of this interface are tables in P4 or maps in eBPF. Other common instances of objects that can be changed at runtime are meters, qdiscs, or stateful registers. The intended behavior of a particular control-plane update is defined in specifications such as P4Runtime [14], Openflow [50], SAI [56], or NETCONF [24].

Our goal. We want to develop a compiler that can specialize network programs given a control-plane configuration. This compiler must be *incremental* with respect to the control plane: The compiler must support automatic respecialization whenever control-plane configurations change, without incurring substantial time on every update—given that most updates do not affect program implementation. We do not consider traffic profiles when specializing because traffic may change more rapidly than control-plane configurations [46].

Why an incremental, specializing compiler? Even though control-plane updates occur less frequently than packet arrivals in the data plane, they do change from time to time, often in response to external events like routing changes, and often in bursts. At the same time, most control-plane updates do not require recompilation of the specialized program because they do not change program semantics. Existing specializing tools such as Morpheus [51], Pipeleon [75], or ESwitch [52] approach this problem by either introducing resource-consuming fall-back datapaths or recompiling the data-plane program every time the control-plane issues an update. When control-plane updates arrive in bursts of hundreds of rules in a few seconds [33, 39–41], recompiling a network program from scratch, which can take several tens of seconds (Tbl. 1), is too slow for a specializing compiler to keep up. Even more recent incremental recompilation approaches require on the order of seconds to complete recompilation [19, 28, 58, 74]. A specializing compiler that is unable to quickly distinguish between a trivial update that doesn't need recompilation (e.g., adding a new NAT entry) and a major data-plane change (e.g., enabling an IPv6 ACL table) will be stuck constantly respecializing.

Our insight. In any network program, we can distinguish runtime-dependent variables into two types: the *data-plane variable*, which depends on data-plane input (e.g., variables

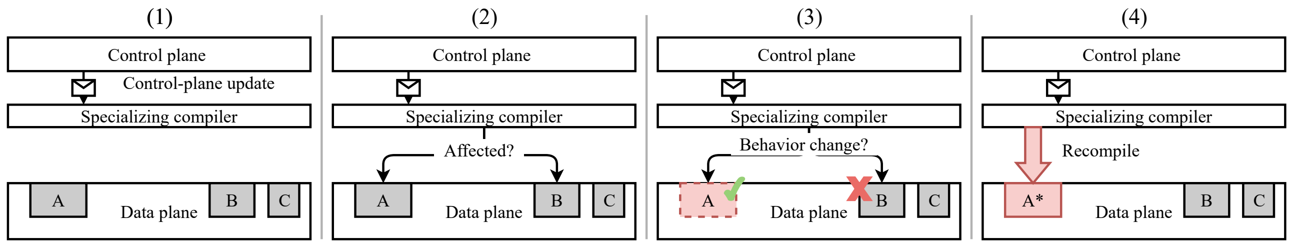


Figure 2: Control-plane-triggered, incremental specialization. Letters describe objects configurable by the control plane.

parsed from a packet header), and the *control-plane variable*, which depends on control-plane input (e.g., an ACL entry which decides whether a packet is forwarded or dropped). For example, in P4, data-plane variables are sourced from the packet through parser extraction calls, whereas control-plane variables are stored in tables and stateful registers. In eBPF on the other hand, data-plane variables are sourced from reads of the packet metadata structure (e.g., `sk_buff`), and control-plane variables are stored in maps (e.g., BPF maps). An incoming packet results in a concrete assignment to the data-plane variables in the program. Similarly, a control-plane update results in an assignment to a subset of control-plane variables.

Any control-plane update can be directly mapped to a component in the data plane (e.g., a table, register, or map). We can use this mapping to implement an incremental compiler. Not every control-plane update introduces a semantic change. Many control-plane entries just increase the likelihood for an already existing data-plane program path to be taken. This allows us to implement a form of taint tracking which lets us quickly identify the affected components. Restrictions in networking DSLs such as a lack of pointer-based indirection, unbounded loops, or jumps make taint tracking tractable. With a taint-tracking system in place, we only need to check whether a particular component's behavior has changed given an update. The way we compute these behavioral semantics, identify affected components, and check quickly whether a change is necessary depends on the particular incremental specialization technique we use. We show a concrete example in §4.

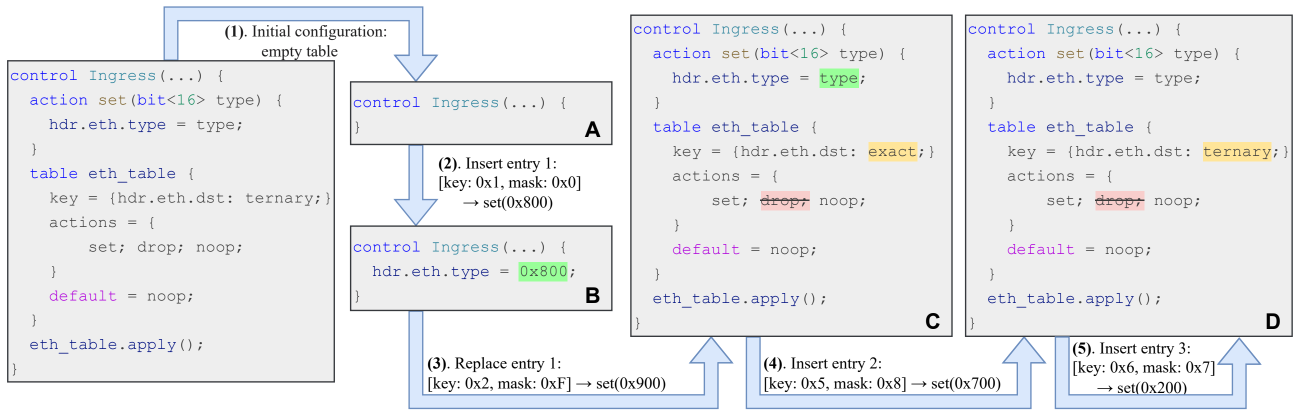
A Control-Plane-Triggered Compiler. Fig. 2 shows a sketch of our proposed approach. (1) The control-plane-triggered compiler is intended to be invoked on every control-plane update and provides feedback on whether a control-plane update requires recompilation. (2) Once a new update is sent to the compiler, it identifies the affected program components based on the control-plane variables "tainted" by the control-plane update. (3) After identifying the affected components, the compiler checks whether the semantics of those data-plane components change. (4) For components that do not need changes, the compiler will forward the update to the device. If the compiler's query indicates that the behavior of a component in the data-plane program will change, the compiler needs to recompile that component before the control-plane update can be installed onto the device. This recompilation (if needed) is done by the device-specific compiler.

3 Specialization Use Cases

Control-plane-triggered specialization as described in §2 can improve resource usage across different network devices. We outline several kinds of specialization use cases.

Resource savings over a program's lifetime. On RMT-style pipelines with hard constraints on the number of computation units, tables, and stateful memories, we can substantially save on hardware resources by specializing to control-plane configurations. As an example, Fig. 3 describes how the implementation of a single P4 table can change in response to different control-plane updates. Initially, the table is empty and can be removed entirely (impl. A). We then insert a single entry, a ternary match with a 0 mask that executes `set(0x800)` as its action. Here, we can inline the table action and save the cost of a table lookup. We then replace the existing entry with a ternary match that uses the full mask (impl. B). This is effectively an exact match entry. Because there is no other entry in the table, we can change the match type of the key, saving Ternary Content Addressable Memory (TCAM) resources. Once we insert entry 2, the table must be implemented as a ternary table (impl. C). The last entry (3) does not change the behavior of the table, and hence no recompilation is needed. Note that, in both implementations C and D, the unused `drop` action is removed from the table, freeing up computation units.

Parser specializations. Several specializations are also possible for parsers in network programs. The `parse_break` command in NPL [9] temporarily suspends the parser to perform table lookups. If the accessed table is empty, we can remove entire parse branches that depend on this particular lookup. P4 Parser value sets (PVS) [15, §13.11] serve a similar function. We can free the TCAMs and SRAMs on a PVS that is not configured. Another network-program-specific specialization is parser-tail pruning. Once we have specialized the


(1). Initial configuration: empty table
(2). Insert entry 1: [key: 0x1, mask: 0x0] → set(0x800)
(3). Replace entry 1: [key: 0x2, mask: 0xF] → set(0x900)
(4). Insert entry 2: [key: 0x5, mask: 0x8] → set(0x700)
(5). Insert entry 3: [key: 0x6, mask: 0x7] → set(0x200)
A
B
C
D

```
control Ingress(...) {
  action set(bit<16> type) {
    hdr.eth.type = type;
  }
  table eth_table {
    key = {hdr.eth.dst: ternary;}
    actions = {
        set; drop; noop;
    }
    default = noop;
  }
  eth_table.apply();
}
```

```
control Ingress(...) {
}
```

```
control Ingress(...) {
  hdr.eth.type = 0x800;
}
```

```
control Ingress(...) {
  action set(bit<16> type) {
    hdr.eth.type = type;
  }
  table eth_table {
    key = {hdr.eth.dst: exact;}
    actions = {
        set; ~~drop;~~ noop;
    }
    default = noop;
  }
  eth_table.apply();
}
```

```
control Ingress(...) {
  action set(bit<16> type) {
    hdr.eth.type = type;
  }
  table eth_table {
    key = {hdr.eth.dst: ternary;}
    actions = {
        set; ~~drop;~~ noop;
    }
    default = noop;
  }
  eth_table.apply();
}
```

Figure 3: For the program on the left, we show control-plane updates 1–5 and their effect on data path implementation.

program, we can check whether the parser itself is doing unnecessary work. Any header at the tail of the parser that is not accessed in the program could be classified as payload. Reducing the amount of parse calls can reduce PHV usage in Tofino or improve packet-processing latency in OvS [53].

Savings in other hardware resources. One, the Tofino programmable switch supports the use of action profiles to support actions (e.g., setting a packet's output port metadata) that are shared among tables. If an action profile is empty, an incremental compiler can specialize the implementation of all tables associated with this action profile. Two, we can specialize device-specific functions. Consider a hardware unit that computes a checksum on a set of headers. Further, let's assume that one of these headers H is set as part of some table action A in table T—as opposed to being parsed out of an incoming packet. If there is no control-plane entry for T that uses A as its action, we know that H is invalid, and hence the checksum will also be invalid, allowing us to directly compute the checksum result, and saving us a checksum unit. Third, if a header is only written by one action and this particular action does not exist in the control-plane configuration, we can simply remove the header in the RMT pipeline to free up packet-header-vector (PHV) resources.

Specializing packet-classification. We can specialize data structures used in the data plane to classify packets based on the actual patterns present in the active control-plane configuration. Often, these techniques involve choosing a less expensive data structure for the given network device. For example, a common, but expensive data structure to classify packets is the TCAM. TCAMs allow matching on header fields based on bitmasks. If we can tell from the current control-plane configuration that only few or no masks at all are necessary, we can replace the TCAM with a simpler matching data structure, e.g., a Semi-TCAM (STCAM) in AMD devices [5]. ESwitch [52] and Morpheus [51] have shown how we can apply similar specializations to software packet-processing devices, such as Open vSwitch (OVS) [55] and eBPF, respectively. NeuroCuts [47] and NuevoMatch [60] train neural networks for more efficient packet classification by mapping a control-plane configuration to an efficient lookup data structure.

How incremental compilation could help. In all of the use cases below, knowledge of the currently active control-plane configurations can help a compiler specialize the underlying implementation of the data-plane program. Further, if we had an incremental compiler [63], it could localize the compiler's effort to specific aspects of the data-plane program. For instance, in Fig. 3, all of the control-plane updates (and hence all of the specializations) pertain to the implementation of a single table, allowing the incremental compiler to ignore the rest of the data-plane program. Similarly, if parser compilation were treated independently of the rest of the program, we could specialize the parser separately in response to which headers are accessed by control-plane entries. Finally, in the context of packet classification, the control-plane update tells us which specific table's implementation to focus on, permitting us to specialize that alone.

4 Feasibility Study

As a preliminary feasibility study, we built Flay. Flay implements incremental partial evaluation for P4 programs. Partial evaluation [42] is a program optimization technique which specializes a program by treating some inputs as constants. Flay specializes P4 programs subject to their control-plane configuration by continuously reoptimizing the running P4 program based on incoming control-plane updates. We picked partial evaluation because, simply by eliminating newly dead code and inlining constants based on the current control-plane configs, we can already implement many of the resource-saving specializations discussed in §3. Flay supports P4 program specialization for various targets (BMv2 [7], Tofino [16], or Xilinx Versal [4]). We also believe

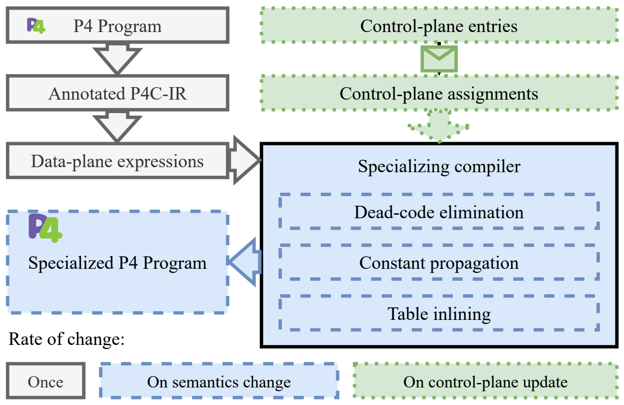
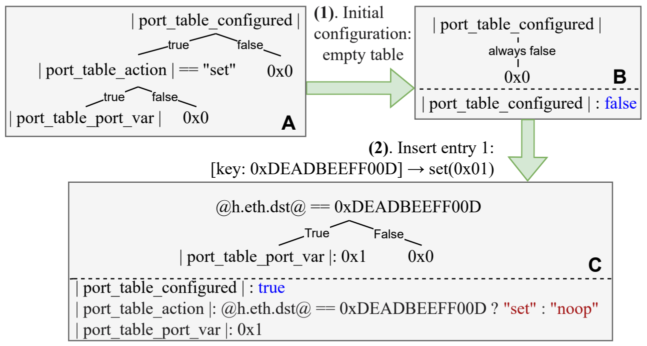


Figure 4: Flay's design.

```
1  control Ingress(...) {
2    action set(bit<9> port_var) {
3      egress_port = port_var;
4    }
5    table port_table {
6      key = {h.eth.dst: exact;}
7      actions = {set; noop;}
8    }
9    apply {
10     egress_port = 0;
11     port_table.apply();
12     h.eth.dst = egress_port == 0 ? 0xAAAAAAAAAAAA : 0xBBBBBBBBBBBB
13   }
14 }
15 # Symbolic value of egress_port variable after executing a line:
16 # Line 9: @egress_port@
17 # Line 10: 0
18 # Line 11: |port_table_configured| && |port_table_action| == "set" ?
19 #          |port_table_var| : 0
20 # Line 12: *unchanged*
```

(a) P4 program setting a port variable based on a table entry.


| port_table_configured |
true
false
| port_table_action | == "set"
0x0
true
false
| port_table_port_var |
0x0
A
(1). Initial configuration: empty table
| port_table_configured |
always false
0x0
B
| port_table_configured | : false
(2). Insert entry 1: [key: 0xDEADBEEFF00D] → set(0x01)
@h.eth.dst@ == 0xDEADBEEFF00D
True
False
| port_table_port_var |: 0x1
0x0
C
| port_table_configured | : true
| port_table_action |: @h.eth.dst@ == 0xDEADBEEFF00D ? "set" : "noop"
| port_table_port_var |: 0x1

(b) Value of `egress_port` at line 12 after each entry update.

Figure 5: Flay's representation of `egress_port`. |x| denotes a control-plane symbol; @x@ a data-plane symbol. Entries below the dotted line are the active control-plane assignments.

that Flay can generalize to packet-processing environments such as restricted C for eBPF [30], NPL [9], or microcode [78]. Flay is available at https://github.com/nyu-systems/flay.

4.1 Flay Overview

Flay implements incremental specialization by representing a network program as a combination of data-plane expressions and control-plane assignments. We can ask specialization queries by substituting the control-plane assignments into placeholders within the data-plane expressions. Fig. 4 shows the high-level workflow of Flay.

Data-plane expressions. We use a simple data-flow analysis coupled with state-merging [6, §5.6] to generate data-plane expressions. For any input program, Flay first computes the data-plane semantics of the program and annotates program points of interest (e.g., if-statements, match-action table execution, map lookups, or variable assignments) with a data-plane expression. The control-plane variables within the expressions act as a placeholder and are later substituted with control-plane assignments. Data-plane variables can assume any value since we do not specialize based on traffic profiles. Our state-merging approach makes any program point annotation *hermetic*, i.e., we can evaluate queries on each annotated program point independently. Lines 15–20 in Fig. 5a demonstrate how we use state-merging to annotate each program line with a snapshot of the value of `egress_port`. If the table does not have a control-plane entry, `port_table_configured` is `false` and `egress_port` will evaluate to `0`. Hence, we can simplify the assignment on line 12 to `h.eth.dst = 0xAAAAAAAAAAAA`. The type of specialization we use influences the number of program points (and hence the work required on each control-plane update). For dead-code elimination we may just want to annotate if-statements, but for constant substitution we may need to annotate any variable read

Control-plane assignments. We represent control-plane entries as a set of control-plane variable assignments. This representation implements the semantics of the control plane as described by the appropriate specification (e.g., P4Runtime). For example, entries that are duplicate or eclipsed by higher-priority entries (and thus have no effect) are omitted in the set of control-plane assignments. To infer the initial assignment set for any configurable data-plane element we consult the device specification. Flay maintains a map which associates a control-plane variable with the set of program points it can influence. On each control-plane update, Flay retrieves all the affected program points from this mapping. For any affected program point, Flay substitutes the control-plane assignments into the expressions associated with the point.

Specialization queries. Once Flay has combined the gathered data-plane expressions with the initial control-plane assignments it specializes the program by asking queries on the joint representation. Typically, the query indicates that the value of the expression has not changed and Flay will forward the update directly to the network device without triggering recompilation. If any program point indicates a change in behavior, Flay must trigger the reoptimization process for the affected data-plane components. We currently ask two types of queries using Flay: 1) Is this particular piece of code executable and 2) Can we replace this program variable with a constant? Concretely, we remove unnecessary table dependencies by deleting unused actions, inline P4 tables which always execute the same action, simplify extern calls, and replace variables and conditions with constants. Flay then passes the specialized program to the device-specific compiler, which optimizes it further. We evaluate some of the benefits of these incremental specializations in §4.2.

An example. Fig. 5b shows how Flay uses a constant propagation query to compute the value of `egress_port` at line 12. The data-plane model in Block `A` represents all the possible values `egress_port` can assume at this line. After obtaining this general representation, we specialize it using the initial control-plane assignment (Block `B`). The control-plane specification for this device prescribes that an empty table executes the default action, which does nothing here. Hence, the assignment sets `port_table_configured` to `false`, which causes `egress_port` to be 0. After receiving an update, we can match on a key field and execute the `set(0x01)` action, which sets `egress_port` to 1 (Block `C`). There are now two possible outcomes for the value of `egress_port`, 0 or 1.

Processing updates quickly. Since, once computed, data-plane expressions do not change, Flay performs extensive preprocessing on expressions to quickly compute queries after control-plane updates. Preprocessing increases initial analysis time but greatly reduces query time. (1) Flay reduces the expression complexity by applying constant folding, common subexpression elimination, and strength reduction. (2) Flay converts each data-plane expression into a representation that supports fast incremental checking specialized towards the particular query. For example, for efficient expression substitution we use Z3's [21] e-matching [20] implementation. Instead of e-matching, we could also use an incremental Datalog evaluation API such as Souffle [62].

Currently, Flay does not support incrementality well in scenarios where tables with complex match keys have many control-plane entries. We show an example of Flay's performance degradation in such scenarios in §4.2. The cause is an inefficient control-plane representation. Since we model the potential matches of an incoming key against all table entries as a single and deeply nested expression, complex keys coupled with large tables can produce a very large expression. Substituting such a complex key expression into a data-plane annotation and checking whether the annotation resolves to a constant can be slow. To make reasonably fast decisions we compromise on Flay's sensitivity. Once a certain threshold of entries (e.g., 100) has been reached, we overapproximate: we assume the entries in the table

P4 Program	Program statements	Compile time	Data-plane analysis time	Update analysis time
scion[22]	582	38s	2s	90ms
switch [66]	786	106s	9s	90ms
middleblock [2]	346	2s	0.6s	5ms
dash [69]	509	2s	1.5s	12ms

Table 2: Flay evaluation times for P4 programs. Compilation is from scratch. Flay's data-plane analysis step runs once and skips the parser. At runtime, Flay only runs update analysis.

cover all its possible actions and action parameters. For example, in Fig 5b, overapproximation would assign `*any*` to `port_table_action` and `port_table_port_var`, which would cause the computed value of `egress_port` to revert to the model shown in Block `A`.

In practice, crossing the threshold rarely requires respecialization because tables with many entries likely cover most of their possible paths already. Drawing from prior techniques [27, 33, 77], we are developing our own compact control-plane representation to speed up update processing with complex control-plane configurations.

4.2 Evaluating Flay

We evaluate how Flay specializes the SCION [22] border router P4 programs written for the Tofino 2 [17] switch. We chose the SCION programs for evaluation because, next to being moderately complex (~1700 LoC), they are supplied with representative control-plane configurations. We use this program to answer questions on specialization, incrementality support, and analysis time.

Can specialization save resources? First, we compile the SCION program without applying Flay's specialization. The program requires the maximum number of Tofino 2 stages. We then specialize the SCION program using the supplied configuration. This configuration does not use IPv6 and all the IPv6 program paths are unused. After removing these unused paths, the program requires 20% fewer stages.

What is the cost of initial data-plane analysis? Our state-merging data-plane analysis is sensitive to programs with many control-flow statements [49]. The initial pass through the program is cheap, but the generated data-plane expressions can become deeply nested. Preprocessing expressions for incrementality support can quickly become slow. To accelerate processing for large programs (e.g., `switch.p4`) we added an option to skip parser analysis. Since Flay's specializations focus on constructs in the control, skipping the parser has little impact on their effectiveness. We evaluate Flay's complexity on a suite of sample programs. The increase is exponential in terms of the number of control paths. Nevertheless, even for large, complex programs, we can complete our initial analysis within a few minutes (Tbl. 2).

What influences Flay's update processing speed? We use a fuzzer [70] to generate 1000 unique IPv4 entries and insert

Total updates installed	Analysis time for 1 incoming update	
	Precise	Overapproximated (>100 entries)
1	~1ms	–
10	~5ms	–
100	~100ms	~1ms
1000	~4000ms	~1ms
10000	~265319ms	~1ms

Table 3: Influence of installed updates on Flay's update processing times for middleblock.p4 [2].

the entries into the SCION IPv4 forwarding table to test how Flay handles a burst of semantics-preserving updates. Flay can determine within a second that the batch of updates does not require program recompilation. We then send a batch of updates that enables the previously unused IPv6 paths in the SCION program. Flay determines respecialization is necessary and triggers the recompilation process. After recompilation, the SCION program requires the maximum number of stages again because all program paths are used. Tbl. 2 shows that Flay is not very sensitive to program complexity. While the time required to process updates increases with program complexity, it generally stays below 100 ms.

Conversely, as discussed in §4, Flay slows down when a complex table has many entries. An example of such a table is the Pre-Ingress ACL table of Google's Middleblock P4 switch model [2]. To characterize the slowdown, we initialize this ACL table with varying number of entries, then send a single update and measure how much time Flay requires to make a decision. Tbl. 3 shows the results. The precise update implementation, which evaluates all entries, already takes 100 ms at only 100 installed entries. Once we overapproximate the entries, update processing time becomes low again.

5 Related Work

Incremental computation [59] is a mature field with a wide range of applications [13, 42, 43, 64, 65]. Recently, work on JIT compilers [73] and feedback-directed optimization [11] articulated the need for incremental specialization.

For network programs, many specialization frameworks use either packet traces or control-plane configurations as input. We classify these into frameworks which specialize network programs once before deployment (offline) and frameworks that continuously specialize the program (online).

Offline-specialization tools. In the context of P4, P5 [1] proposes control-plane-based optimization to simplify dependencies between P4 tables. P2GO [72] is a profile-guided specialization tool where a profile is the combination of a packet trace and the expected control-plane configuration. Parasol [35] uses traffic profiles to generate data structures optimized to that profile. NFReducer [23] and PacketMill [25] specialize network functions chains by applying a series of framework (e.g., ClickNF [44]) optimizations based on initial control-plane configurations. mSwitch [38] inlines switching rules within the VALE [61] software switch. Relative to these tools, our approach is to specialize *continuously*, in response to every control-plane update.

Online-specialization tools. Bhatia et al. [8] specialize the Linux network stack by inlining installed IPv4 routes and bridging route changes using a NAT until respecialization has completed. ESwitch [52] and Hoda [53] continuously specialize OvS. ESwitch optimizes OvS by changing packet-matching templates based on user-supplied traffic and flow entry patterns. Hoda instead produces a new, specialized parser and megaflow cache from existing cache rules. Pipeleon is a profile-guided specialization framework targeting P4 SmartNICs [75]. Morpheus [51] performs profile-guided optimization for eBPF. To deal with input profile changes, these tools respecialize on each control-plane update or periodically trigger recompilation. Our approach can defer recompilation until program semantics change.

6 Conclusion and Future Outlook

This paper argues for control-plane-triggered and incremental compilation as a new way of thinking about compilers for packet processing. While our initial results are encouraging, we see at least three broad areas where work is necessary.

First, while we make every attempt to avoid recompilation on as many control-plane updates as possible, we eventually have to recompile when a control-plane update triggers a change in semantics. In such cases, recompilation times should ideally be low, but, currently, we are at the mercy of device-specific compilers that treat the whole program as a monolithic unit to be compiled from scratch. Recent work on modularity in network programming languages [26, 68, 71] and hardware support for partial reconfiguration [71, 74, 79] points the way towards recompilation of just the modules (such as specific tables) that have changed. Second, we plan to use Flay as a vehicle to explore tradeoffs between specialization time and specialization quality to further decrease recompile times in response to control-plane updates. Third, we plan to extend these ideas to other programmable network ecosystems such as eBPF/XDP [36], DPDK-based systems [31, 34, 55, 57], or SmartNICs [4, 18, 54].

7 Acknowledgements

We would like to thank the anonymous HotNets reviewers for their encouraging feedback. We are also grateful to Nate Foster and Liangcheng Yu for their insightful comments on paper drafts and the project. This work is supported in part by an NSF CAREER award (2340748), NSF grant 2008048, 2145471, and a Google Cyber NYC Research Award. This work was also partially supported by the European Union - Next Generation EU under the Italian National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.3, CUP D43C22003080001, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART").

References

[1] Anubhavnidhi Abhashkumar, Jeongkeun Lee, Jean Tourrilhes, Sujata Banerjee, Wenfei Wu, Joon-Myung Kang, and Aditya Akella. P5: Policy-driven optimization of P4 pipeline. In *ACM SOSR*, 2017.

[2] Kinan Dak Albab, Jonathan Dilorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Tirmazi, Jiaqi Gao, and Minlan Yu. SwitchV: Automated SDN switch validation with P4 models. In *ACM SIGCOMM*, 2022.

[3] Albert Gran Alcoz, Martin Strohmeier, Vincent Lenders, and Laurent Vanbever. Aggregate-based congestion control for pulse-wave DDoS defense. In *ACM SIGCOMM*, 2022.

[4] AMD. AMD versal adaptive SoCs. https://www.amd.com/en/products/adaptive-socs-and-fpgas/versal.html. Accessed: 2024-10-22.

[5] AMD. Content addressable memory (CAM). https://www.xilinx.com/products/intellectual-property/ef-di-cam.html. Accessed: 2024-10-22.

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 2018.

[7] Antonin Bas. The reference P4 software switch. https://github.com/p4lang/behavioral-model, 2014. Accessed: 2024-10-22.

[8] Sapan Bhatia, Charles Consel, A-F Le Meur, and Calton Pu. Automatic specialization of protocol stacks in operating system kernels. In *29th Annual IEEE International Conference on Local Computer Networks*, 2004.

[9] Broadcom. NPL: Open, high-level language for developing feature-rich solutions for programmable networking platforms. https://nplang.org/, 2019. Accessed: 2024-10-22.

[10] Mihai Budiu and Chris Dodd. The $P4_{16}$ programming language. *ACM SIGOPS Operating Systems Review*, 2017.

[11] Dehao Chen, David Xinliang Li, and Tipp Moseley. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, 2016.

[12] 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*, 2020.

[13] Charles Consel, Julia L Lawall, and Anne-Françoise Le Meur. A tour of Tempo: a program specializer for the c language. *Science of computer programming*, 2004.

[14] The P4.org consortium. The P4Runtime specification, version 1.3.0. https://p4.org/p4-spec/p4runtime/v1.3.0/P4Runtime-Spec.html, December 2020.

[15] The P4.org consortium. The $P4_{16}$ language specification, version 1.2.4. https://p4.org/p4-spec/docs/P4-16-v1.2.4.html, May 2023.

[16] Intel Corporation. Industry-first co-packaged optics Ethernet switch. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html. Accessed: 2024-10-22.

[17] Intel Corporation. Second-generation P4-programmable Ethernet switch ASIC that continues to deliver programmability without compromise. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html. Accessed: 2024-10-22.

[18] Intel Corporation. The infrastructure processing unit (IPU). https://www.intel.de/content/www/de/de/products/network-io/smartnic.html, 2022. Accessed: 2024-10-22.

[19] Rajdeep Das and Alex C Snoeren. Memory management in ActiveRMT: Towards runtime-programmable switches. In *ACM SIGCOMM*, 2023.

[20] Leonardo De Moura and Nikolaj Bjørner. Efficient e-matching for SMT solvers. In *CADE-21: 21st International Conference on Automated Deduction*, 2007.

[21] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[22] Joeri de Ruiter and Caspar Schutijser. Next-generation internet at terabit speed: SCION in P4. In *ACM CoNEXT*, 2021.

[23] Bangwen Deng, Wenfei Wu, and Linhai Song. Redundant logic elimination in network functions. In *ACM SOSR*, 2020.

[24] Rob Enns. NETCONF configuration protocol (RFC 4741). IETF Request For Comments, 2006.

[25] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Packetmill: toward per-core 100-gbps networking. In *ACM ASPLOS*, 2021.

[26] Ali Fattaholmanan, Mario Baldi, Antonio Carzaniga, and Robert Soulé. P4 weaver: Supporting modular and incremental programming in P4. In *ACM SOSR*, 2021.

[27] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *USENIX OSDI*, 2016.

[28] Yong Feng, Zhikang Chen, Haoyu Song, Wenquan Xu, Jiahao Li, Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu. Enabling in-situ programmability in network data plane: From architecture to language. In *USENIX NSDI*, 2022.

[29] The Linux Foundation. middleblock.p4. https://github.com/sonic-net/sonic-pins/blob/main/sai_p4/instantiations/google/middleblock.p4, 2021. Accessed: 2024-10-22.

[30] The Linux Foundation. eBPF: Introduction, tutorials & community resources. https://ebpf.io/, 2022. Accessed: 2024-10-22.

[31] Massimo Gallo and Rafael Laufer. ClickNF: a modular stack for custom network functions. In *USENIX ATC*, 2018.

[32] Xiangyu Gao, Divya Raghunathan, Ruijie Fang, Tao Wang, Xiaotong Zhu, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Cat: A solver-aided compiler for packet-processing pipelines. In *ACM ASPLOS*, 2023.

[33] Dong Guo, Shenshen Chen, Kai Gao, Qiao Xiang, Ying Zhang, and Y Richard Yang. Flash: fast, consistent data plane verification for large-scale network settings. In *ACM SIGCOMM*, 2022.

[34] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A software NIC to augment hardware. Technical report, University of California at Berkeley, 2015.

[35] Mary Hogan, Devon Loehr, John Sonchack, Shir Landau Feibish, Jennifer Rexford, and David Walker. Automated optimization of parameterized data-plane programs with Parasol. *arXiv preprint arXiv:2402.11155*, 2024.

[36] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The EXpress Data Path: Fast programmable packet processing in the operating system kernel. In *ACM CoNEXT*, 2018.

[37] Thomas Holterbach, Stefano Vissicchio, Alberto Dainotti, and Laurent Vanbever. SWIFT: Predictive fast reroute. In *ACM SIGCOMM*, 2017.

[38] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. mswitch: a highly-scalable, modular software switch. In *ACM SOSR*, 2015.

[39] Danny Yuxing Huang, Kenneth Yocum, and Alex C Snoeren. High-fidelity switch models for software-defined network emulation. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013.

[40] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM*, 2015.

[41] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. *ACM SIGCOMM Computer Communication Review*, 2014.

[42] Neil D Jones. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 1996.

[43] Neil D Jones and Arne J Glenstrup. Program generation, termination, and binding-time analysis. In *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002 Pittsburgh, PA, USA, October 6–8, 2002 Proceedings*, 2002.

[44] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 2000.

[45] Jonatan Langlet, Ran Ben Basat, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. Direct telemetry access. In *ACM SIGCOMM*, 2023.

[46] Yifan Li, Jiaqi Gao, Ennan Zhai, Mengqi Liu, Kun Liu, and Hongqiang Harry Liu. Cetus: Releasing p4 programmers from the chore of trial and error compiling. In *USENIX NSDI*, 2022.

[47] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. Neural packet classification. In *ACM SIGCOMM*, 2019.

[48] Jinsong Mao, Hailun Ding, Juan Zhai, and Shiqing Ma. Merlin: Multi-tier optimization of eBPF code for performance and compactness. In *ACM ASPLOS*, 2024.

[49] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, 1976.

[50] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 2008.

[51] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. Domain specific run time optimization for software data planes. In *ACM ASPLOS*, 2022.

[52] László Molnár, Gergely Pongrácz, Gábor Enyedi, Zoltán Lajos Kis, Levente Csikor, Ferenc Juhász, Attila Kőrösi, and Gábor Rétvári. Dataplane specialization for high-performance openflow software switching. In *ACM SIGCOMM*, 2016.

[53] Heng Pan, Peng He, Zhenyu Li, Pan Zhang, Junjie Wan, Yuhao Zhou, XiongChun Duan, Yu Zhang, and Gaogang Xie. Hoda: a high-performance Open vSwitch dataplane with multiple specialized data paths. In *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024.

[54] Pensando. A new way of thinking about next-gen cloud architectures. https://p4.org/p4/pensando-joins-p4.html, 2020. Accessed: 2024-10-22.

[55] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *USENIX NSDI*, 2015.

[56] Open Compute Project. SAI: Switch abstraction interface. https://www.opencompute.org/projects/sai. Accessed: 2024-10-22.

[57] LF Projects. Vector packet processing. https://github.com/FDio/vpp/, 2024. Accessed: 2024-10-22.

[58] Yiming Qiu, Ryan Beckett, and Ang Chen. Synthesizing runtime programmable switch updates. In *USENIX NSDI*, 2023.

[59] Ganesan Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1993.

[60] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. Scaling open vswitch with a computational cache. In *USENIX NSDI*, 2022.

[61] Luigi Rizzo and Giuseppe Lettieri. VALE, a switched ethernet for virtual machines. In *ACM CoNEXT*, 2012.

[62] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, 2016.

[63] Mayer D Schwartz, Norman M Delisle, and Vimal S Begwani. Incremental compilation in magpie. *ACM SIGPlan Notices*, 1984.

[64] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[65] Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. Propeller: A profile guided, relinking optimizer for warehouse-scale applications. In *ACM ASPLOS*, 2023.

[66] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. DC.p4: Programming the forwarding plane of a data-center switch. In *ACM SOSR*, 2015.

[67] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *ACM SIGCOMM*, 2021.

[68] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. Composing dataplane programs with μP4. In *ACM SIGCOMM*, 2020.

[69] Reshma Sudarshan and Chris Sommers. P4 as a single source of truth for sonic dash use cases on both softswitch and hardware. https://opennetworking.org/2022-p4-workshop-gated/, 2022. Accessed: 2024-10-22.

[70] New York University. ControlPlaneSmith: Generate control-plane configurations from P4 programs. https://github.com/nyu-systems/rtsmith. Accessed: 2024-10-22.

[71] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. Isolation mechanisms for high-speed packet-processing pipelines. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[72] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. P2GO: P4 profile-guided optimizations. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, 2020.

[73] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[74] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. Runtime programmable switches. In *USENIX NSDI*, 2022.

[75] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Songyuan Sui, Khalid Manaa, Omer Shabtai, Yonatan Piasetzky, Matty Kadosh, Arvind Krishnamurthy, TS Eugene Ng, et al. Unleashing SmartNIC packet processing performance in P4. In *ACM SIGCOMM*, 2023.

[76] Qiongwen Xu, Michael D Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. Synthesizing safe and efficient kernel extensions for packet processing. In *ACM SIGCOMM*, 2021.

[77] Hongkun Yang and Simon S Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 2015.

[78] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. Using Trio – Juniper networks' programmable chipset – for emerging in-network applications. In *ACM SIGCOMM*, 2022.

[79] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In *ACM SIGCOMM*, 2020.