Mantis: Reactive Programmable Switches

Liangcheng Yu
University of Pennsylvania
leoyu@seas.upenn.edu

John Sonchack
Princeton University
jsonch@princeton.edu

Vincent Liu
University of Pennsylvania
liuv@seas.upenn.edu

ABSTRACT

For modern data center switches, the ability to—with minimum latency and maximum flexibility—*react* to current network conditions is important for managing increasingly dynamic networks. The traditional approach to implementing this type of behavior is through a control plane that is orders of magnitude slower than the speed at which typical data center congestion events occur. More recent alternatives like programmable switches can remember statistics about passing traffic and adjust behavior accordingly, but unfortunately, their capabilities severely limit what can be done.

In this paper, we present Mantis, a framework for implementing fine-grained reactive behavior on today's programmable switches with the help of a specialized reactive control plane architecture. Mantis is, thus, a combination of language for specifying dynamic components of packet processing and an optimized, general, and safe control loop for implementing them. Mantis provides a simple-to-reason-about set of abstractions for users, and the Mantis control plane can react to changes in the network in 10s of µs.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **Programming interfaces**; In-network processing; Network dynamics;

KEYWORDS

Programmable networks, P4, Reconfiguration, Control plane

ACM Reference Format:
Liangcheng Yu, John Sonchack, and Vincent Liu. 2020. Mantis: Reactive Programmable Switches. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3387514.3405870

1 INTRODUCTION

Modern data center networks are becoming increasingly dynamic. Their switches, in addition to providing simple forwarding functionality, are often expected to change their packet processing behavior in reaction to fluctuating network conditions, e.g., to distribute traffic [2, 11, 17, 49], handle failures [27, 28], implement flow control [14, 32, 50], or apply expressive security policies [20, 25, 45].

SIGCOMM '20, August 10–14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-7955-7/20/08…$15.00
https://doi.org/10.1145/3387514.3405870

```
malleable value value_var { width : 16; init : 1; }
malleable field field_var {
    width : 32; init : hdr.foo;
    alts {hdr.foo, hdr.bar}
}
malleable table table_var {
    reads { ${field_var} : exact; }
    actions { my_action; drop; }
}
action my_action() {
    add(${field_var}, hdr.baz, ${value_var});
}
reaction my_reaction(reg qdepths[1:10]) {
    uint16_t current_max = 0, max_port = 0;
    for (int i = 1; i <= 10; ++i)
        if (qdepths[i] > current_max) {
            current_max = qdepths[i]; max_port = i;
        }
    ${value_var} = max_port;
}
```

Figure 1: An example P4R code snippet with fields, values, and a table that can be modified at runtime using fine-grained reactions. `malleable` variables are annotated as such. Malleable field and value variables are referenced as `${var}`.

For each of these tasks, reacting to the current state of the network is critical to maintaining strict Service-Level Objectives (SLOs).

Parallel to this trend has been a realization that the majority of congestion events in today's data centers are microscopic in duration. For instance, [57] found that, in one production data center, 90% of continuous periods of high utilization lasted for less than 200 µs. Studies of other data centers have shown similar levels of volatility in network traffic over small timescales [5, 37].

As a result, recent work has proposed pushing an increasing amount of adaptive behavior into the network devices themselves. Load balancing is one such example. While switches have long been able to statically spread load over the network using mechanisms like ECMP, microbursts and other transient events provide a compelling case for making more complex routing decisions locally at each device, where it is possible to react at very small timescales [2, 11, 50]. This is in contrast to more traditional OpenFlow-style approaches, which rely on a relatively slow control loop passing through a centralized controller. Similarly reactive systems have been proposed for other use cases [20, 25, 28, 52].

The cost of faster reaction time in these systems? For many, it is custom hardware modifications that add the features directly into the data plane [2, 11, 28]. Unfortunately, developing these custom ASICs is both extremely expensive and time-consuming. Programmable switches provide a promising alternative, allowing users to integrate some amount of statistics gathering and computation into the packet processing pipeline, but as we describe in Section 2, the limitations of today's programmable switches are well known and difficult to overcome, despite sustained efforts from the networking community.

In this paper, we present Mantis, a framework for implementing fine-grained reactive behavior in today's programmable switches. Like traditional network architectures, Mantis relies on the data plane to perform packet processing and the control plane to implement arbitrary control logic. Unlike traditional architectures, the Mantis control plane is designed to—at the granularity of 10s of µs—continually measure and adjust the behavior of the data plane. Mantis is, thus, a combination of (1) an extension to the P4 language, P4R, that helps to specify which parts of the data plane should be malleable, and (2) the Mantis agent, an optimized control plane that provides both a Turing-complete substrate and serializability guarantees for user-defined reactions. While Mantis's reliance on the CPU means that it cannot react to *every* packet, it enables sub-RTT reaction time, which we show is sufficient for many applications.

Figure 1 shows an example P4R program. In it, we can observe a set of novel primitives that can replace any of their traditional P4 counterparts: malleable *values*, which can be reconfigured at runtime to take on any numeric value; malleable *fields*, which can be reconfigured to reference P4 packet/metadata fields; and malleable *tables*, which function as normal match-action tables, but are augmented with support for fast and serializable updates. Mantis will continuously poll headers/state from the data plane and modify the above primitives based on user-specified *reactions*—each iteration of the reaction loop allows arbitrarily complex reaction logic, is guaranteed to operate on fresh data, permits concurrent traditional control plane operations, and provides serializable isolation with respect to reads, writes, and packet processing. Our prototype and evaluation of Mantis and the P4R language demonstrate their utility in a wide range of use cases that are difficult/expensive to implement otherwise.[1] This paper makes the following contributions:

- We introduce a novel extension to the P4 language, P4R, that treats *reactions* to current network conditions as a first-class citizen. Along with this language, we present a Flex/Bison-based compiler that translates P4R into a pair of artifacts: (1) a valid but malleable P4-14 v1.0.5 program and (2) C reaction code that polls data plane state and updates its malleable portions.
- We also develop the Mantis agent, a control plane architecture that can execute reactions quickly and safely on a Wedge100BF-series switch. Depending on the reaction complexity, our current implementation can react at a granularity of 10s of µs (less than an RTT in most networks) and guarantees serializable isolation of both the measurements and updates.
- Finally, we present a series of use cases that demonstrate the utility of Mantis and dynamic reaction. A surprising result of our work is that, not only does Mantis outperform centralized approaches, it can often outperform pure data plane approaches along important metrics.

This work does not raise any ethical issues.

2 BACKGROUND AND MOTIVATION

We begin this paper by describing the architecture of today's Reconfigurable Match Table (RMT) switches with a focus on their capacity to react to current network conditions. To that end, RMT switches are based on the abstraction of a pipeline of match-action tables. For a given packet and table, the switch will index into the table using a subset of packets' fields and metadata, extracting an action that it will then apply to the current packet. Some switches may also include a small amount of SRAM for stateful processing. The 'reconfigurable' of RMT refers to the ability to change both the fields considered in the match as well as the action that is executed.

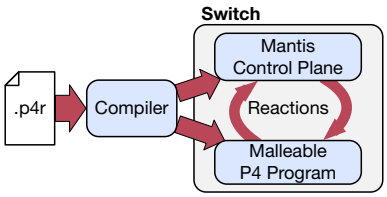


Figure 2: Mantis. A P4R program is compiled into a pair of artifacts that support high-frequency, switch-local reactions.

In this context, a *reaction* involves aggregating statistics (e.g., packet count or queue depth) from across packets and then using those statistics to influence the processing of subsequent packets (e.g., by redirecting a subset of them or tagging them with a computed value). In principle, an RMT switch with stateful SRAM can be configured to do both of the above actions—measurement and control—entirely within the data plane, and prior work has done exactly this for a subset of use cases [12, 40]. In practice, however, today's state-of-the-art RMT implementations suffer from a number of well-known limitations, some of which may be fundamental to efficient ASICs [4] and none of which are addressed by existing hardware. These include, but are not limited to constraints on the operations allowed in actions (e.g., no multiplication/division, limited branching, etc.), a fixed number of stages in the pipeline, restrictions of SRAM accesses to a single element/stage, and a disconnect between ingress/egress pipelines.

When encountering one of these limitations, prior work has tended to take one of a few different approaches. Some expend heroic efforts approximate the original algorithms in a way that fits the constraints [39, 40, 53]. Others assume novel hardware primitives that add the appropriate flexibility to the data plane [15, 22, 33, 41]. Some in the latter category might still be developed into the former; however, for a general approach that works on today's networks, workarounds typically involve one of the following.

Resubmission and recirculation. The most direct way to circumvent the data plane limitations is to send traffic back through the packet processing pipeline multiple times and, if necessary, across pipelines [4, 15, 22, 41]. Theoretically, with enough recirculations, one can overcome limitations in both computational power (achieving Turing-completeness) and memory flexibility (acquiring access to any number of SRAM entries any number of times).

The primary drawback is that, each time a packet is recirculated, it potentially impacts other traffic. Recirculating every packet twice, for instance, drops usable throughput of the switch to 38%; three times reduces throughput to just 16% [51]. As modern switches are generally limited by their packet-level bandwidth, the size of the recirculated packets is immaterial. Recirculated packet processing also introduces the potential for violations of consistency/isolation.

Chaining/reloading the data plane. For cases where stage count is the main limitation, an alternative approach is to chain or swap in/out network functionality as needed [24, 52]. Unfortunately,

[1] The open-source P4R compiler can be found at `https://github.com/eniac/Mantis`

chaining imposes requirements on the network topology and workload, and installing a new data plane program renders the switch temporarily unavailable (several seconds in current implementations). It, therefore, only applies for coarse-grained reactions and cases where sufficient capacity exists elsewhere in the network.

Control plane assistance. Finally, we note that data planes have long been unable to implement all of the functionality needed inside a network. Instead, they are typically augmented with a control plane an onboard CPU to which the data plane can offload more complex tasks such as routing and management when those types of packets arrive. The two planes communicate by passing messages or, from the control plane, by polling counters and updating table entries—all of these can be done without disrupting normal switch operations. Unfortunately, traditional control planes assume that accesses are not time-critical and, thus, are generally orders of magnitude slower than the duration of network events [11, 28].

3 DESIGN OVERVIEW

Mantis is a framework for fast, expressive in-network reactions on today's RMT switches. Mantis has two primary goals:

(1) To enable general and flexible dynamic reactions that surpass the limitations of today's switches—measuring an arbitrary set of data plane metrics, computing an arbitrary set of statistics over that data, and manipulating the data plane without impacting normal traffic. To sustain typical network volatility, this process should be able to operate on a sub-RTT granularity.
(2) To package the above capability into a reaction abstraction that hides the many complexities of implementing reactive behavior, e.g., ensuring that the data plane is malleable, coordinating data-control plane communication at runtime, and reasoning about asynchronous behavior.

Explicitly *not* a goal of our system is support for arbitrary changes to the data plane at runtime. For that, we refer interested readers to prior work that has successfully emulated P4 using match-action tables [13], albeit at a high cost (for an L2 switch, a 6.5× increase in match stages and 83% bandwidth penalty). Instead, we assume that the general structure of the data plane is known a priori and that reactions only need to touch a subset of data plane objects.

We demonstrate, using Mantis, that the above approach is sufficient to implement a range of network architectures that are difficult and/or costly to implement in today's RMT switches. We discuss and evaluate these applications, listed in Table 1, in Section 8.

Core abstractions. Two abstractions underlie our work:

Malleable Entities — In Mantis, specific primitives in the data plane program can be tagged as 'malleable,' indicating that they should be amenable to fine-grained modification at runtime. Malleable values, used in the expressions of data-plane actions, can take on any constant value; malleable fields act as dynamic references to a predefined set of existing header and metadata fields; and malleable tables function exactly like normal match-action tables, but with the ability to be modified at a fine-granularity.

Reactions — Measuring the network and modifying malleable portions of the data plane are 'reactions'—small C functions that are compiled and dynamically loaded into a custom, reaction-centric control plane running on each switch's local CPU. In Mantis, the control plane will, as quickly as possible, poll the parameters of each reaction function and react to the measurements by updating malleable entities according to these user-defined functions.

⟨*p4_declaration*⟩ ::= ⟨*mbl_declaration*⟩ | ⟨*reaction_declaration*⟩ | ...

⟨*mbl_declaration*⟩ ::= 'malleable' ⟨*table_declaration*⟩ (malleable entities)
| 'malleable' ⟨*mbl_val_declaration*⟩
| 'malleable' ⟨*mbl_fld_declaration*⟩

⟨*mbl_val_declaration*⟩ ::= 'value' ⟨*mbl_name*⟩ '{'
⟨*width_declaration*⟩ ';'
'init :' ⟨*const_value*⟩ ';' '}'

⟨*mbl_fld_declaration*⟩ ::= 'field' ⟨*mbl_name*⟩ '{'
⟨*width_declaration*⟩ ';'
'init :' ⟨*field_ref*⟩ ';'
'alts {' ⟨*field_ref*⟩ [',' ⟨*field_ref*⟩]* '}' ';' '}'

⟨*reaction_declaration*⟩ ::= 'reaction' ⟨*reaction_name*⟩ (reactions)
'(' [⟨*reaction_args*⟩ [, ⟨*reaction_args*⟩]*] ')'
'{ // C-like code }'

⟨*reaction_args*⟩ ::= 'ing' ⟨*reaction_arg*⟩
| 'egr' ⟨*reaction_arg*⟩
| 'reg' ⟨*register_ref*⟩ '[' ⟨*const_value*⟩ ':' ⟨*const_value*⟩ ']'

⟨*reaction_arg*⟩ ::= '${' ⟨*mbl_read_ref*⟩ '}'
| ⟨*field_ref*⟩
| ⟨*header_ref*⟩
| ⟨*field_value*⟩

⟨*field_or_masked_ref*⟩ ::= '${' ⟨*mbl_read_ref*⟩ '}' (references)
| '${' ⟨*mbl_read_ref*⟩ '}' 'mask' ⟨*const_value*⟩
| ...

⟨*arg*⟩ ::= '${' ⟨*mbl_read_ref*⟩ '}' | ...

⟨*exp*⟩ ::= '${' ⟨*mbl_read_ref*⟩ '}' | ...

Figure 3: The P4R extensions to the P4-14 v1.0.5 grammar. Gray non-terminal nodes refer to legacy rules in [48], and nodes ending in *_name* indicate strings whose first character is a letter. *mbl_read_ref*s can access both malleable values and fields. Note that all writes in P4-14 are done via primitive actions, which we omit for similar reasons to [48].

System architecture. On top of the above abstractions, Mantis combines a language, a compiler, and a control plane architecture, all designed to enable fast, simple, and safe control loops for programmable switches. Figure 2 depicts the architecture of Mantis.

- *The P4R language:* Actualizing our two core abstractions is a simple extension to P4—one where certain fields, values, and tables can be tagged as 'malleable.'
- *The Mantis compiler:* The compiler transforms P4R programs into a pair of artifacts: (1) a valid P4 program that reformulates the P4R to ensure that metrics are exported and that `malleables` are runtime-configurable and (2) a reaction function implementation that interfaces with the generated P4 program.
- *The Mantis control plane:* An optimized control plane agent running on the switch CPU is responsible for the rapid, serializable coordination of measurement and updates.

Paper roadmap. We start in Section 4 by describing the P4R language and how Mantis translates from P4R to P4 *without* any isolation guarantees. Section 5 then introduces Mantis's approach for guaranteeing per-pipeline serializable isolation of reads, writes, and packet processing. Finally, Section 6 presents the Mantis control plane before delving into the implementation/evaluation.

4 LANGUAGE AND TRANSFORMATIONS

Adhering to best practices in language design [21], P4R reuses the basic syntax and semantics of the P4 programming language. It then allows users to tag various P4 objects as 'malleable' and define the reaction functions that modify those objects. We already saw an example of both language features in Figure 1.

Grammar. Briefly, malleable fields and values are declared with a width, an initial value, and in the case of a malleable field, a set of potential aliases to which the entity can reference. Malleable tables are declared with an annotation to indicate that the compiler should prepare for its use in a reaction loop. Otherwise, all three can be used in the same way as their traditional P4 counterparts.

The precise extensions we make to the P4-14 grammar are shown in Figure 3. The grammar follows the naming conventions of [48]. Note that, like [48], we omit our changes to primitive actions such as `modify_field` and `add_to_field`, whose existence is platform dependent. In general, however, any field or value parameter to these primitive actions may be replaced with a reference to a `malleable` (*mbl_read_ref* or *mbl_write_ref*, depending on the semantics).

Reaction functions. Of note are reaction functions like the one in Figure 1 that allow users to embed C code that specifies the control plane behavior that accompanies the data plane implementation.

Syntactically, reactions mirror C, but with a couple of changes. The first is the parameters to a reaction, which are a set of fields, registers, or malleable fields/values from the data plane. Before executing the body of the reaction, Mantis polls the current value of all of these parameters. Note that if there are multiple line cards with distinct register state, a separate instance of the Mantis agent will run for each. The second is the use of `malleables` within the function body. For malleable fields and values, these can be referenced with the same `${var}` notation used in the rest of the P4R program—the compiler will replace them with generated functions that write to the data plane or read the last written value, depending on the context. For malleable tables, users can interact directly via a set of automatically generated library functions, e.g., `table_var.addEntry(...)`.

Semantically, all registered reaction functions are executed sequentially, in a loop. Mantis does not guarantee a specific ordering but does guarantee serializability between parameter polling, updates to malleable entities, and packets' processing (see Section 5).

4.1 Producing Malleable P4

Supporting fast and safe reactions is Mantis's compiler, which transforms P4R into a valid P4 program, but one in which malleable entities can be rapidly updated at runtime. In this section, we describe through several examples the necessary transformations for malleable fields and values without considering isolation guarantees. Note that we do not describe one-off writes of malleable tables as (ignoring isolation) they are already modifiable in today's switches.

Values. Figure 4 shows a simple example of a definition and use of a malleable value, `value_var`. The original P4R code can be found in the four *non*-bolded lines, which contain the entity definition and its use within the P4 `add` primitive.

The Mantis compiler transforms the code as follows. It instantiates the value in a metadata header (`p4r_meta_`) and generates

```
- malleable value value_var { width : 16; init : 1; }
+ header_type p4r_meta_t_ {
+     fields { value_var : 16; }
+ }
+ metadata p4r_meta_t_ p4r_meta_ { value_var : 1; };

  // Applied once at the beginning of the pipeline
+ table p4r_init_ {
+     actions { p4r_init_action_; }
+     size : 1;
+ }
+ action p4r_init_action_(value_var) {
+     modify_field(p4r_meta_.value_var, value_var);
+ }
  action my_action() {
±     add(hdr.foo, hdr.bar, ${value_var} p4r_meta_.value_var);
  }
```

Figure 4: Mantis's transformation of a malleable value. ~~Strikethroughs~~ and '-' annotations indicate P4R code that is removed by the transformation; bold text and '+' annotations indicate P4 code that is generated by Mantis.

```
- malleable field write_var {
-     width : 32; init : hdr.foo;
-     alts { hdr.foo, hdr.bar }
- }
+ header_type p4r_meta_t_ {
+     fields { write_var_alt : 1; }
+ }
+ metadata p4r_meta_t_ p4r_meta_;
  // Action applied once (with value loads)
+ action p4r_init_action_(write_var_alt) {
+     modify_field(p4r_meta_.write_var_alt, write_var_alt);
+ }
  // For every use of the malleable field
  table my_table {
      reads { hdr.qux : exact;
±             p4r_meta_.write_var_alt : exact; }
±     actions { my_action_hdr_foo_;
±               my_action_hdr_bar_; }
  }
± action my_action_hdr_foo_(baz) {
±     modify_field(${write_var} hdr.foo, baz);
  }
+ action my_action_hdr_bar_(baz) {
+     modify_field(hdr.bar, baz);
+ }
```

Figure 5: Transformation for malleable fields that we wish to use on the 'left-hand side' of assignments.

an associated table (`p4r_init_`) with a single possible action. This table is applied at the beginning of each packet processing pipeline, and it is what allows Mantis to assign different values to the `malleable` at runtime by updating just a single table entry. As we will see later, this initialization table serves many purposes, configuring `malleables` and version control bits for the entire pipeline.

Fields (write). P4R also includes malleable fields, which act as references to a predefined set of existing P4 fields; users can dynamically 'shift' the target of the reference to any member of the set, e.g., to change the matched field of a table. As references, malleable fields are L-values, meaning that they can appear on either the left- or right-hand side of an assignment operator in the data plane program. We focus first on 'left-hand' usages. Figure 5 shows an example. Specifically, it shows a scenario where the programmer seeks to store the value of `baz` into either `hdr.foo` or `hdr.bar`.

A naïve implementation of this functionality would be to replace the `malleable` with a generated metadata field and, after every use

```
- malleable field read_var {
-    width : 32; init : hdr.foo;
-    alts { hdr.foo, hdr.bar }
- }
+ header_type p4r_meta_t_ {
+    fields { read_var_alt : 1; }
+ }
+ metadata p4r_meta_t_ p4r_meta_;
  // Action applied once (with value loads)
+ action p4r_init_action_(read_var_alt) {
+     modify_field(p4r_meta_.read_var_alt, read_var_alt);
+ }
  // For every use of the malleable field
  table my_table {
±    reads { ${read_var} : exact;
+            hdr.foo : ternary; hdr.bar : ternary;
±            p4r_meta_.read_var_alt : exact; }
±    actions { my_action_hdr_foo_;
±              my_action_hdr_bar_; }
  }
± action my_action_hdr_foo_() {
±    add(hdr.qux, hdr.baz, ${read_var} hdr.foo);
  }
+ action my_action_hdr_bar_() {
+     add(hdr.qux, hdr.baz, hdr.bar);
+ }
```

Figure 6: Transformation for malleable fields that we wish to use on the 'right-hand side' of assignments. This example combines uses inside an action and a table match field.

of it, add a match-action table whose sole purpose is to copy the current value of the generated field back into the referenced field. The inserted table would have a distinct action for every possible 'alt,' and users would modify the default action when changing the target of the reference. Unfortunately, there are several issues with this strawman. First, it adds additional tables and potentially stages to the data plane program. Second, it violates the atomicity of reference shifts as a concurrent shift might cause the reference to act as hdr.foo or hdr.bar in different actions applied to the same packet. Even without concurrent shifts, uses of both the malleable and the field to which it references in the same action can be problematic.

To address the above challenges, Mantis performs two tasks. The first is to declare and load, at the beginning of the pipeline and for every relevant malleable field, a metadata field (e.g., write_var_alt) with width $\lceil \log_2 |alts| \rceil$ that determines, at runtime, the alternative that it references. The second is to transform every table that assigns the malleable field to also match on write_var_alt.

This extra match field allows the data plane to call specialized action functions that are instantiated for each possible configuration of the malleable fields. While this strategy increases the number of entries in affected tables to:

$$\sum_{(m,a)\in Entries} \left(\prod_{v\in \text{mbls}(a)} |v_{alts}| \right)$$

it avoids the table/stage costs of the strawman (often the bottleneck in programmable switches) and the atomicity issues. We anticipate that the number of affected actions, the number of malleable fields per action, and the number of alternatives per malleable field will all be relatively small in most cases.

Fields (read). Malleable fields can also be used on the right-hand side of assignments almost anywhere a field can be referenced in P4, e.g., inside an action, as a table match field, or in a field_list. Figure 6 shows a couple of examples.

Inside an action, we can apply the previous method of loading the selector field at the beginning of the pipeline and specializing actions. Slightly more complex is the use of malleable fields in table matches. Here, the compiler, in addition to matching on read_var_alt, replaces the malleable match field with |alts| instantiated match fields. For example, when the user adds an entry for ${read_var} = 0, Mantis inserts two entries into my_table:

- (foo=0, bar=*, read_var_alt=0)
- (foo=*, bar=0, read_var_alt=1)

Note that this means exact matches on a malleable field need to become ternary to accommodate the wildcard; ternary and lpm matches can remain. Also note that using a malleable in a table match, on its own, does not necessitate action specialization—specialization is only necessary if it is used within the given action.

Compound usages. While the above examples all include only a single malleable entity, Mantis allows the use of multiple entities in the same program and the same tables/actions.

One place where multiple malleables would interact is the initialization at the beginning of the pipeline. To minimize the number of necessary tables and actions, we can reuse a single init_action for multiple malleables (field or value) by passing in multiple parameters and including multiple assignments in the action body. If the number or aggregate size of the parameters exceeds the limits of a single action, Mantis will create multiple init tables. In this case, minimizing the number of tables involves a bin packing problem. Mantis solves this with a simple greedy algorithm in which it sorts the parameters in order of decreasing size and finds the 'first fit.'

The other place where they might interact is in the tables and actions of the P4 program. For malleable values, their composition is trivial as any instance can be directly replaced with the designated metadata field regardless of context. For malleable fields, multiple uses of the same field—whether left-hand or right—can be coalesced; each action needs to be specialized at most one time. For uses of different fields, transformations are applied recursively. For example, two malleable fields used in the same action will require two stages of action specialization that will result in an enumeration of all possible permutation of alternatives. Note an optimization when the fields are read, but not written—loading values in prior stages may result in lower overhead than instantiating all permutations.

4.2 Gathering Measurements

In addition to ensuring that portions of the data plane can be rapidly updated, Mantis also ensures that reaction function parameters can be rapidly read. While P4 provides many ways to read information from the data plane (e.g., digests, counters, the copy_to_cpu flag, etc.), to ensure fast reaction time, the chosen mechanisms need to have the following properties:

R1 Measurement should *not* be on a per-packet basis. While we seek fast reaction time, switch CPUs are not equipped and should not be expected to handle line-rate processing.

R2 The measurement schedule should be flexible. While reactions should be as close to real-time as possible, concurrent management tasks and varying reaction execution time mean that the Mantis should tolerate fluctuating measurement intervals.

R3 Measurements should return the most recent data. For instance, the regular export of digests from the data plane would be undesirable as the most recent digests might be head-of-line blocked behind previously unprocessed digests.

Mantis presents an abstraction where the data plane updates measurements on *every* packet, but the control plane only polls those measurements when it is ready to process the next iteration of the reaction loop. Mantis implements this protocol via stateful registers that can be updated in the data plane and polled from the control plane; Mantis stores the polled values in a C variable/array for use in the user-defined reactions. Note that, this pull-based model will only see a subset of updates, thus, users should ensure that any necessary information is retained across packets.

Compiler transformations. Header/metadata reaction parameters are collected from every passing packet into generated registers at the end of the pipeline specified by their `ing`/`egr` annotation; the Mantis compiler places the register after the last modification to the field. User-defined register parameters can be read from the control plane directly, modulo the transformations described in Section 5.2.

Similar to the generation of the init action, Mantis packs header and metadata reaction parameters into as few registers as possible using the sorted-first-fit algorithm discussed previously. The only difference is that parameters from different reactions may be considered separately when packing. Although this may consume more resources than otherwise, it allows Mantis to poll only the most relevant parameters immediately before executing the reaction, which improves the freshness of the measured data.

5 ENFORCING ISOLATION[2]

A critical piece of the reaction abstraction is isolation between measurement, modifications, and packet processing. To see why this is important, consider a reaction function that takes as arguments the 5-tuple from a packet. While a user might reasonably expect that the parameters passed into her reaction function all came from a single packet, without isolation, this is unlikely unless no new packets arrive between the first and last measurement.

To address this challenge, *Mantis provides per-pipeline, per-reaction serializable isolation between measurements, malleable entity updates, and packet processing*. Said differently, from the perspective of a single packet processing pipeline, the three types of operations—gathering of measurements, application of a reaction, and processing of packets—all appear to execute in some sequential order, despite the inherent parallelism of packet processing.

This particular level of isolation is deliberate as it is both practical and efficient to implement. Stronger guarantees like grouping measurement and updates into a single transaction are useful but difficult to implement in today's switches. Similarly, cross-pipeline guarantees, while potentially useful, would require some type of in-band coordination between all pipelines [53]. We leave an exploration of these stronger models for future work.

5.1 Serializable Isolation of Updates

We begin with how Mantis guarantees serializability of reactions' effects before discussing measurement collection in Section 5.2.

[2]'Isolation' here refers to same type of isolation used in ACID [55].

5.1.1 *Updates to fields and values*

For malleable fields and values, the generated P4 of Section 4.1 is specifically crafted to be atomically modifiable. In particular, for both types of entities, their value is determined at the beginning of each pipeline, in the `p4r_init_` table. As RMT switches typically guarantee the consistency of a single table entry modification, so long as we can pack all configuration of malleable entities into a single `p4r_init_action_` (see Section 4.1), we can leverage the action as a serialization point. As a concrete example, consider a P4R program with two `malleables` (`value_var` and `field_var`):

```
action p4r_init_action_(value_var, field_var_alt) {
  modify_field(p4r_meta_.value_var, value_var);
  modify_field(p4r_meta_.field_var_alt, field_var_alt);
}
```

A single table entry update can change both atomically. New packets that enter the pipeline will use the updated assignments, while packets that have already passed this stage will continue to use the previous set of assignments.

The above strategy works until the P4R metadata used in a single pipeline of the P4R program exceeds the allowed size of the action (a platform-dependent value). As mentioned in Section 4.1, this case forces the compiler to split the `p4r_init_` table into several, e.g., `p4r_init1_`, `p4r_init2_`, etc. Updates to these tables can be made serializable by treating all except the first as normal, malleable tables and using the method described in the subsequent section.

5.1.2 *Updates to tables*

Handling malleable table modifications is slightly more complex. Conceptually, Mantis's approach is similar to that of [35, 36], but with a few critical differences that stem from Mantis's goal of extremely fast and repeated updates.

More specifically, in [35], Reitblatt et al. guarantee consistent updates in SDN deployments using a two-phase protocol. The protocol assumes that every packet is tagged with the current version number, i. Thus, to install a new configuration, the first step is to add the complete set of new rules across the internal nodes of the network such that the new rules only match on packets with version $i + 1$. The second step is then to, one-by-one, update all ingress nodes to tag packets entering the network with version $i + 1$. After a conservative timeout, the older configuration set is eventually removed from the internal nodes.

There are at least a couple of issues with applying the above protocol directly to Mantis's reaction loop. The first relates to the handling of frequent updates: given how often Mantis can update tables, conservative timeouts and the need to keep around multiple 'in-flight' updates can easily lead to order-of-magnitude increases in necessary table space. The second relates to reaction time: every update in [35] requires an insertion for every table entry in the new configuration, regardless of whether it was changed from the previous version or not.[3] Removal of stale versions doubles the latency overhead when the throughput of the control plane is the bottleneck.

Mantis, in contrast, guarantees serializability of groups of arbitrary and repeated table updates, where the required time is

[3]While [35] mentions possible optimizations that only apply the delta between old and new configurations, it does not discuss how to handle more than one such update.

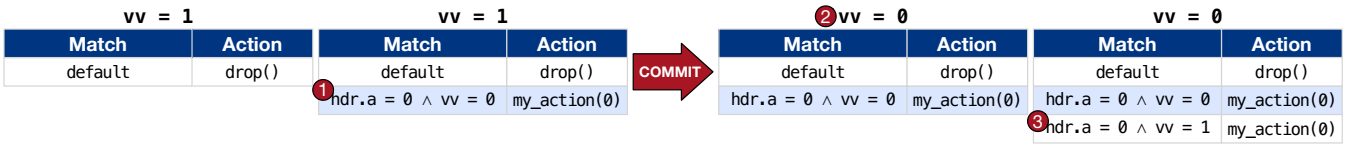


Figure 7: Ensuring sequentially consistency for table entry adds using three-phase updates. Multiple entries across multiple tables can be modified in step ❶ before they are atomically committed in step ❷. The rule is mirrored in step ❸ to assist with fast subsequent updates.

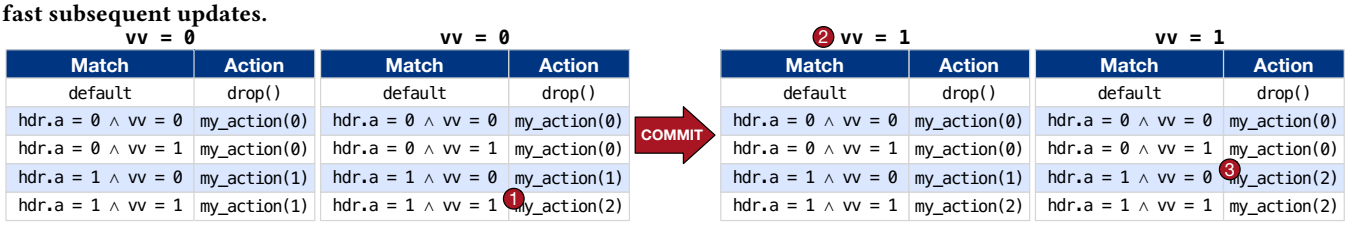


Figure 8: Ensuring sequentially consistency for table entry updates using three-phase updates. As in Figure 7, multiple entries and tables can be modified in step ❶. Unaffected entries remain untouched.

proportional to the number of P4R table interactions and the space overhead is bounded. Mantis's approach relies on a 1-bit version control flag, vv, that is set in the `init_action` alongside the malleable fields/values; vv is also added as an exact-match field to every malleable table. With the vv field, every entry in every malleable table is doubled: one copy with vv = 0 and the other with vv = 1. Active entries can be flipped atomically by updating the vv bit.

Note that a 1-bit version flag is sufficient in Mantis as Mantis loads malleable entity configurations and the version control bit at the beginning of each pipeline and deliberately does not guarantee cross-pipeline isolation (e.g., between ingress and egress or across recirculations). Thus, old versions only persist for the maximum latency of a pipeline, which is typically measured in the 100s of nanoseconds. PCIe latency from the control plane is an order of magnitude higher, so the maximum number of active versions is two, regardless of the complexity of the reaction's effects.

Adding a new entry to the table at runtime employs a three-step update, as shown in Figure 7. Assume, w.l.o.g., that vv begins at 1. In this configuration, the entries with vv = 1 serve as the primary copy, while the entries with vv = 0 serve as a shadow copy.

(1) The Mantis control plane first *prepares* the entries it wishes to add by adding them to the table, but with the requirement that vv = 0. Any number of entries and tables can be modified in this step; meanwhile, all packets will continue to use the vv = 1 and the `default` action.
(2) In the second step, Mantis then *commits* all of the added entries by atomically flipping the version control bit, vv = vv ⊕ 1 by updating the `p4r_init_` table. Note that any inflight packets that have already passed the `init` stage will continue to use the vv = 1 copy even after the commit.
(3) Finally, so that the entry can withstand a subsequent flip back to vv = 1, Mantis *mirrors* updates to the shadow copy. While this step has no visible effect on the network, it amortizes the cost of maintaining the shadow to keep latency more predictable.

Updating an existing table entry proceeds as in Figure 8. As mentioned above, init tables beyond the first are handled using the same mechanism. `p4r_init1_` (the first table) is considered the 'master' and contains the version control bit; all other init tables will contain two entries (one for each version) just like a malleable

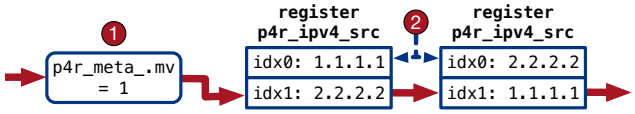


Figure 9: Ensuring measurement isolation for measured data plane fields. For `mv` = 1, index 1 is the working copy and index 0 is the checkpoint copy. For `mv` = 0, vice versa.

table. Thus, when dealing with multiple init tables, the master is always updated last. Deleting an entry looks similar to adding a table entry, but in reverse: the shadow copy is deleted in the prepare step and the original primary is deleted after the commit. All three types of operations—add, update, and remove—can share a prepare and commit step, even if they touch the same tables and entries. A proof of serializability of this process follows from that of [35].

5.2 Serializable Isolation of Measurements

Mantis also guarantees that the polling of the registers that store reaction parameters reflects a serial execution with respect to packet processing. As mentioned, a naïve implementation of register polling would result in an inconsistent view of reaction arguments.

Fields. Mantis ensures that the data plane will not overwrite the registers that store reaction-parameter fields (see Section 4.2) by using a register array rather than an individual register. These arrays have two entries each: a 'working' copy and a 'checkpoint' copy, both gated on a 1-bit `mv` bit that is set in the `p4r_init_` action of each pipeline along with vv. We configure the data plane such that it only writes to the working copy.

Figure 9 demonstrates an example usage of this mechanism. When the control plane wishes to measure a group of generated field-storing registers, it first ❶ flips the measurement version bit. Assuming that the flip was from 0 → 1, index 0 of both registers are now the checkpoint copies and should not be touched by the data plane. Mantis can, therefore, take as much time as it needs to ❷ read those values; meanwhile, the data plane will continue to update the working-copy entries.

Registers and register arrays. Mantis can also collect values contained in stateful elements, e.g., registers, using a 'double-buffering' scheme. Specifically, it creates a duplicate version of the register with twice as many instances. In every action that writes to the

original user-defined register, Mantis saves the written value (and in the case of a register array, the accessed index) to metadata fields and mirrors the fields to the duplicated register. The written index is the original index prepended by the mv bit.

A complicating factor in register measurement (and the reason why we need a duplicate rather than reusing the original) is that not every register will be updated on every packet. In fact, in the case of a register array, at most a single index will be updated per packet. Because of this, the control plane may observe stale values. For example, consider a case where a register R contains the value r_i in both mv = 0 and mv = 1, and R gets updated to r_{i+1} in the working copy, mv = 1. If the control plane flips mv twice before another update of R, then it will observe r_{i+1} followed by a stale reading of r_i. Until a new update of R, the measured value will alternate between r_i and r_{i+1}.

The above effect necessitates an additional mechanism. Mantis adds to every duplicated register a 'timestamp' register whose entries are incremented every time the associated register's entries are updated. This allows the Mantis control plane to identify which entries have changed since the copy was last read. The control plane keeps a cache of these values; entries are only replaced when the associated timestamp is updated, ensuring that it holds only the most up-to-date contents of every register entry.

We note a potential optimization when the stateful element is never read within the data plane—a common pattern with switch counters and other statistics. In this case, the original register is not necessary and can be eliminated.

6 THE MANTIS CONTROL PLANE

The Mantis control plane runs on a switch's onboard CPUs and uses the measurements and malleable code described in the previous sections to interact with the switching ASIC. Modern data center switches already use this CPU for tasks such as routing, monitoring, and configuration; however, these interactions are traditionally assumed to be one-off and asynchronous, i.e., 'on the slow path.'

We pursue a different goal: to, as quickly as possible, poll data plane registers and react to them in a user-defined fashion. Rather than treating each interaction between the data and control plane as an isolated event, Mantis presents an alternative architecture—one where the control plane executes one of a set of predetermined actions repeatedly, and without pause. With a highly optimized control-plane agent and driver, Mantis can execute iterations of the control plane loop at granularities that are on the same order of magnitude as the PCIe latency of the underlying system, and an order of magnitude lower than a typical data center RTT.

Control plane architecture. The operation of the Mantis control plane is split into two phases:

(1) *Prologue:* The prologue phase encompasses the initialization of malleable values/fields, populating initial table entries, setting up memoization, and configuring driver sessions of the switch.
(2) *Dialogue:* The dialogue phase is where the control plane—as rapidly as possible—polls measurement registers and executes user-defined reactions based on the collected measurement.

Mantis is explicitly optimized for repeated accesses and updates to the same set of reaction parameters and malleables; this includes several custom driver modifications that support repeat interactions. Mantis optimizations include precomputation of metadata during the prologue; batching of requests during the dialogue; and caching/memoization of device instructions in the prologue (for statically computable driver operations) as well as the dialogue (for repeated table modifications). The latter is particularly important for speeding up mv updates, etc. Thus, control flow is as follows:

```
// prologue
helper_state = precompute_metadata();
memo = setup_cache(helper_state);
run_user_initialization(helper_state, memo);
// dialogue
while(!stopped) {
  updateTable(memo, "p4r_init_", {measure_ver : mv ^ 1});
  read_measurements(memo, mv); mv ^= 1;
  run_user_reaction(memo, helper_state, vv ^ 1);
  updateTable(memo, "p4r_init_", {config_ver : vv ^ 1});
  fill_shadow_tables(memo, vv); vv ^= 1;
}
```

The dialogue loop is single-threaded to avoid driver contention and consistency issues; however, if the switch contains multiple disjoint linecards or pipelines, these can be handled by spawning multiple Mantis agent threads, each handling its own component. To minimize latency, Mantis runs as a busy loop in a reserved CPU core, with the option to trade latency for lower CPU utilization.

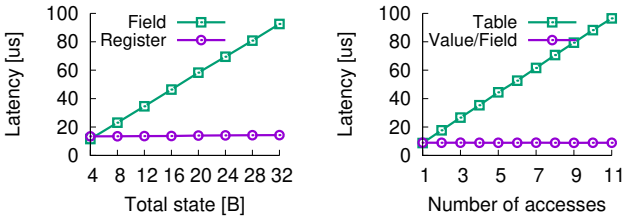
Stateful dialogue. We note that Mantis allows users to retain state across iterations of the dialogue loop. Examples of behavior that may require this functionality include computing average throughput, tracking buffer depth gradients, or using the reaction loop to sample statistics over time. Mantis supports this behavior intrinsically through C static variables, which, when used inside a function, allocate space in the DATA segment of the program's memory and retain their value across function invocations.

Legacy control plane accesses. We also note that Mantis does not preclude legacy control plane accesses, e.g., for routing protocols, handling of higher-level protocols, and manual network operator interaction. Concurrent use is fine as the underlying drivers are typically designed to be thread-safe. Further, due to the poll-based and single-threaded nature of the Mantis agent, at most one reaction is active at any time. Thus, the CPU-ASIC interactions of a legacy application will only need to queue behind at most one set of operations from Mantis. We evaluate this effect in Section 8.2.

7 IMPLEMENTATION

We implemented a prototype of Mantis, including a P4R compiler, control plane agent, and modified driver infrastructure. Our prototype runs on a Wedge100BF-32X.

Compiler. The Mantis compiler translates .p4r files into a P4 program and C library. In total, the compiler implementation consists of 4,000 lines of C++ and around 1,250 lines of grammar. The compiler's parsing frontend is implemented with Flex/Bison. While parsing, the compiler builds an AST of the input program and, over several translation passes, adds/updates nodes to implement the transformations in Sections 4 and 5. While parsing the P4R program, the compiler also extracts reaction function definitions. With the help of the P4 compiler, the compiler translates the arguments and malleable entity modifications into executable code that properly handles argument mirroring and isolation.



(a) Argument measurement. (b) Malleable entity updates.

Figure 10: Latency of raw measurements/updates in Mantis. These numbers do not include isolation mechanisms.

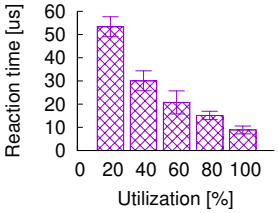


Figure 11: CPU utilization and reaction time tradeoff.

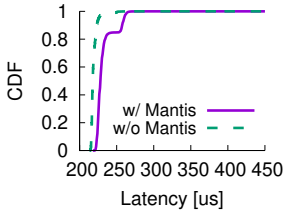


Figure 12: Latency of a concurrent legacy table update with and without Mantis.

Dynamic loading. The C reaction function is compiled into a shared object with `gcc` so that, at runtime, Mantis can load the user reaction loops via shared objects and dynamic loading. Not only does this separate the implementation of the control plane from that of user code, it also potentially enables users to change their reaction functions without interrupting switch operations.

To signal a change, a user-defined signal will activate a transition flag in the running agent. The flag will break out of the reaction loop after any current dialogue completes, unload the previous dialogue module, and link the new shared object. Users can specify whether the prologue user initialization should be re-executed.

Control plane. Our prototype control plane dynamically unloads/reloads `.so` files that implement the reaction prologues and dialogues, before executing the high-frequency measurement and reaction loop described in Section 6. To ensure fast reaction time, we reserve a core for the reaction loop, configure the loop thread with high priority, and set a `SCHED_FIFO` real-time scheduling policy. As mentioned in Section 6, these can be scaled back in return for increased reaction time. We also modify the existing drivers and control plane interfaces in order to optimize latency.

8 EVALUATION

We evaluated Mantis using our prototype implementation. All experiments were run on a hardware testbed consisting of a Wedge100-BF-32X switch connected to a set of servers via 25 Gbps links.

8.1 Mantis Achieves Fast Reaction Times

To measure the reaction time of Mantis, we microbenchmark raw operations (before the mechanisms of Section 5) in Mantis, from which we can construct a cost model. The microbenchmarks show the latency of reading reaction arguments and writing to `malleables`.

Figure 10a plots the latency of measuring the data plane versus the total size of the state that is read. We show results for both 32-bit field arguments and 32-bit register arguments. For field arguments (ingress or egress), the latency of measurement is dependent on the number of packed 32-bit registers that the control plane must read. This value increases linearly modulo packing efficiency. For register arguments, our kernel driver optimizations ensure that reads of multiple entries of a single register array are cheap, each additional byte incurring only 10s of ns of latency. We do not show results for reading from multiple register arrays as the results are identical to field arguments (as field arguments are implemented as registers).

Figure 10b shows results for updates of the data plane. Here, we plot latency versus the number of updates. For scalar `malleables` (fields and values), the latency of updates is constant as long as all of the accesses can be handled within a single `p4r_init_` table (very large in today's switches). After that point, we would need to include the latency of the update protocol. For malleable tables, latency increases linearly with the number of entries modified. Note that, for table insertions, the latency may be more complex [8]; however, we anticipate that most reactions will be updates or involve smaller tables.

The total latency of a reaction function is, thus, approximately:

$$F_{10b}(1\ \text{tblMod}) + \sum_{a \in args} \Big(F_{10a}(a)\Big) + C$$

$$+ \sum_{t \in tblMods} \Big(2F_{10b}(t)\Big) + 2F_{10b}(N_{init} - 1) + F_{10b}(1\ \text{tblMod})$$

where F_{10a} and F_{10b} are functions that correspond to Figures 10a and 10b, respectively; C is the execution time of the reaction logic; and N_{init} is the number of `init` tables in the generated program. The first half of the equation corresponds to the latency of serializable measurement and the reaction logic. The second half corresponds to the latency of serializable updates. For all of the use cases in Table 1, end-to-end reaction time was on the order of 10s of µs.

8.2 Mantis Can Co-exist with Other Functions

We next explore Mantis's overhead in switch and CPU resources.

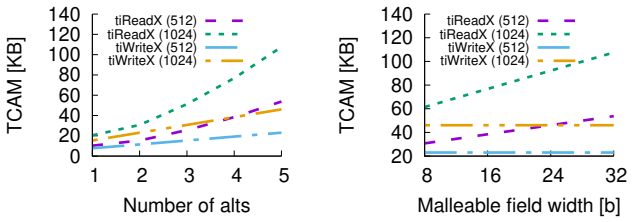
CPU. By default, the Mantis control plane agent occupies one dedicated core for its dialogue loop; however, as mentioned in Section 6, it is possible to reduce this utilization at the cost of slower reaction times. Figure 11 shows this tradeoff for the update of a single malleable field with `nanosleep` for pacing. Reducing utilization to 20% still keeps the average reaction time to 10s of µs.

We also evaluate the impact of Mantis's fast reaction loop on concurrent, legacy switch operations. Specifically, we configure a parallel control plane (running on a different core of the switch CPU) that submits a continuous stream of table entry updates to the switch. We note that this is likely more aggressive than most legacy control planes. Mantis does slow down its neighbor, but the impact is relatively small; it mostly comes when the neighbor's update is blocked behind Mantis's current operation, creating a bimodal distribution. Even so, the median and p99 latency of legacy switch operations in the presence of Mantis versus without it are within 4.64% and 6.45%, respectively.

Memory. The other primary overhead of Mantis is switch memory, which is used for init tables, measurement registers, malleable table shadow entries, and transformations for malleable fields. The effect of the first three sources are simple to reason about: init tables add a small number of tables with 1–2 entries each, measurement

		Malleables			LoC		Control Flow			Memory		
Example	**Reaction**	**val**	**fld**	**tbl**	**P4R**	**P4**	**Stgs**	**Tbls**	**Regs**	**SRAM**	**TCAM**	**Metadata**
Flow size estimation and DoS mitigation	Reads packet headers from the data plane to derive an estimate of the data sent by all senders in the network. Blocks senders that exceed a threshold rate.			✓	81	95	2	2	1	48KB	1.28KB	263b
Route recomputation	Detects failures using a gray failure detector—marking the link as down if received heartbeats dip below $\delta = \lfloor \eta \frac{T_d}{T_s} \rfloor$ for consecutive loops. Routes are recomputed on detection.			✓	30	158	1	6	6	192KB	0KB	160b
Hash polarization mitigation	Reads queue depth of a set of load-balanced ECMP ports. If there is persistent imbalance, changes the ECMP hashing strategy to prevent polarization.		✓		157	245	3	8	1	160KB	0KB	498b
Reinforcement Learning	Reads packet counts and queue depths for different ports to compute a reward function. Uses RL techniques to optimize DCTCP marking threshold reconfiguration policy.	✓			132	239	2	13	6	192KB	0KB	380b

Table 1: Examples of network features that can be formulated as reactions. Evaluation metrics are measured in terms of marginal increase over a basic router. See Section 8 for a more detailed evaluation of these examples.



(a) $K = 32$. **(b)** $A = 5$.

Figure 13: Malleable field TCAM usage.

registers are proportional to the number of arguments, and shadow table entries double the memory required for each malleable table.

The memory cost for malleable field transformations is slightly more complex. To evaluate it, we consider a K-bit malleable field `${X}` with A possible alternatives. We use a table `tiWriteX` that matches on the 5-tuple (all ternary matches) and writes to `${X}` in an action, similar to Figure 5. We also use a table `tiReadX` that uses X in an action *and* as a field match, similar to Figure 6 but matching on the 5-tuple plus X.

Figure 13 shows both tables' TCAM usage (the main bottleneck in this scenario). We evaluate two table occupancies, 512 and 1024. These are the number of user-defined entries, not the number of actual entries, which will be higher to account for the instantiated actions. As shown in Figure 13a, for a given field width, TCAM usage scales linearly with A in `tiWriteX`. For `tiReadX`, usage is asymptotically quadratic because the compiler needs to instantiate actions *and* add A extra ternary match columns. Varying the field width, we obtain Figure 13b, which shows that for `tiReadX`, usage is proportional to K and `tiWriteX` size is constant with respect to K as, when A is fixed, the number of action instantiations is fixed.

8.3 Mantis, in Context

We also evaluate Mantis in the context of the use cases of Table 1. We emphasize that these examples are not complete solutions, nor do they preclude the existence of a future workaround. Rather, they are intended as instruments through which we can understand the utility of Mantis, its relationship to existing data/control-plane alternatives, and the range of what it can express.

8.3.1 *Use Case #1: Flow Size Estimation and DoS Mitigation*

The first use case we examine includes a classic problem in computer networks: flow size estimation. Flow sizes are useful for a variety of tasks, including Heavy Hitters, DDoS victim detection, etc [29]. Unfortunately, given the scale of today's networks, obtaining a precise account of the network's flow sizes is not always feasible. Instead, most modern approaches rely on approximation.

Two solutions are prototypical for this problem. The first is sFlow and its variants [34] where the control plane constructs approximate flow statistics from sampled packets. The second is sketch-based approaches [44] where the data plane records, in a compact representation, statistics over flows. With representatives in both a traditional control plane utility and programmable data plane algorithm, this use case is an ideal proving ground for Mantis.

As a reaction, we use a similar setup to Poseidon [56], which among other things, proposed dynamic reinstallation of data plane programs to respond to DDoS attacks. For simplicity, we model their per-sender statistics and rate-limiting defense, but the same techniques would apply to 5-tuples and more complex defenses.

Algorithm. We configure the data plane to track the current packet's source IP and a counter of the total number of bytes received. The reaction takes these two values as parameters and keeps a hash table of all sources. On every iteration, it attributes the marginal increase in total byte count from the previous dialogue to the given source IP. The reaction then computes the rate using $\frac{\hat{f}_t - \hat{f}_{t_0}}{t - t_0}$, where $\hat{f}_t$ is the counter at time t and t_0 is the time immediately prior to the first observation of the flow. To prevent spurious detection of new flows, we impose a minimum duration before a flow becomes eligible for blocking. For our experiments, we set a simple 1 Gbps threshold, but reiterate that arbitrary C is allowed.

Results. To evaluate the accuracy of size estimation in Mantis, we use `tcpreplay` with a CAIDA [7] ISP-backbone trace. For this experiment, Mantis was able to sustain a sampling rate of ~10 µs, corresponding to an average of ~1 in 5 packets.

Figure 14 shows the average estimation error of Mantis versus the sFlow-based estimator and a pair of data plane implementations. For sFlow, we use the 1:30,000 sampling frequency suggested in [37].

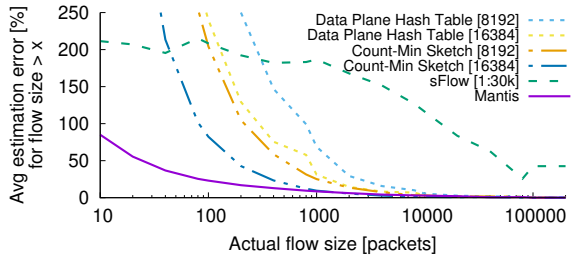


Figure 14: Average estimation error for Mantis and several alternatives. Mantis outperforms sFlow by orders of magnitude, and when equalizing the number of stages, beats data plane implementations for small flows as well.

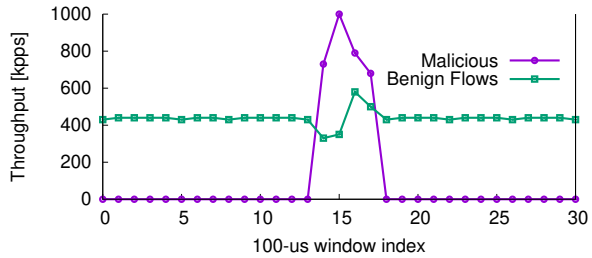


Figure 15: Aggregate throughput for legitimate flows from $S_{\{1,2,\ldots,250\}}$ sending to D. When S_0 begins to flood the network, Mantis detects and suppresses it orders of magnitude faster than similar systems (cf. Figure 14 of [56]).

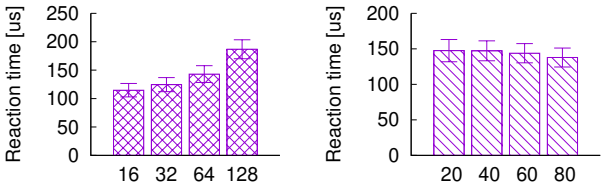
For the data plane implementations, we include a hash table as well as a 2-stage count-min sketch. Following the configuration guidance of [44], we chunk the trace into 20 s blocks (each with around 8.9 M packets and 370 K flows) and configure all tables to have 8,192 entries (the next power of two above their setting of 4,500). We also show data plane results for 16 K entries; Mantis's performance was unchanged.

Compared to sFlow, Mantis is significantly more accurate due to its higher sampling frequency. This effect becomes pronounced as we approach sFlow's sampling granularity. Compared to both data-plane approaches, Mantis provides slightly worse but comparable accuracy for large flows, and orders of magnitude better accuracy for small flows. The overall trend holds across table sizes. The reason is that, in Mantis, inaccuracy is caused primarily by sampling error, which is bounded; in contrast, sketch inaccuracy is due to collisions, which may misattribute arbitrarily many bytes to the wrong flow.

Figure 15 shows the reaction in action. 250 legitimate TCP flows utilize 20% of a 10 Gbps bottleneck link before a single malicious sender arrives and blasts UDP traffic at 25 Gbps using a DPDK sending script. The Mantis reaction can install a mitigation rule within ~100 µs (from the timestamp of the first packet of the malicious flow). Accounting for packet delays and TCP mechanisms, the benign flows return to steady-state operation within ~500 µs, orders of magnitude faster than traditional reconfiguration.

8.3.2 *Use Case #2: Route Recomputation on Gray-failures*

The second use case leverages Mantis's reaction time more directly via a gray-failure route recomputation scheme. In this use case, the reaction loop measures the frequency of heartbeats from neighboring nodes and triggers a control-plane route recomputation when the frequency drops below a threshold.



(a) Latency vs ports (η = 0.5) (b) Latency vs η (ports = 64)

Figure 16: The time to accurately detect failures and reroute for a robust Mantis-based gray failure detector.

Algorithm. Our failure detection scheme is based on a previously proposed gray-failure detector [28] whose original design required specialized hardware support. In our formulation, we install in every node adjacent to the switch a heartbeat generator that produces high-priority packets at a granularity of T_s (1 µs in our tests). The detecting switch accumulates a per-port count of these heartbeats and the current timestamp in the data plane.

By polling (serializably) the counts and timestamp, a Mantis reaction can compare the number of observed heartbeat messages with the number of *expected* messages. More specifically, it can use a threshold $\delta = \lfloor \eta \frac{T_d}{T_s} \rfloor$ where T_d is the time since the last dialogue and $\eta \in [0, 1]$ captures expectations for the successful delivery of heartbeats—a high η will demand a more reliable link and catch failures faster and a low η will allow for more outliers at the cost of reaction time. Two consecutive polling periods with fewer than δ heartbeats trigger recomputation and installation of new routes.

A few features of this use case would be challenging without Mantis. Compared to a traditional control plane solution that polls raw packet counters, Mantis offers fast reaction speed and serializable reads of counters/timestamps that remove inaccuracies due to data/control-plane latency. Compared to a fully data plane solution that computes the threshold and activates detours, Mantis avoids the significant overheads of approximating division when computing δ [39] by offloading it to the control plane. Involving the control plane also opens up the possibility of arbitrary route recomputation and table modifications (data planes are typically limited to static backup paths or inefficient detour protocols [27]).

Results. Figure 16 shows the end-to-end reaction time of failure detection and route recomputation in Mantis for various configuration parameters. To emulate link failures, we leveraged switch APIs that disabled physical ports on the switch. Reaction time is defined as the difference between the control-plane timestamps of the link-down event and the installation of the new routing rules.

Figure 16a shows that Mantis can restore connectivity within 100–200 µs with low variance. What variance it does have is a result of the position of the failure in the first T_d window—if it occurs right before the ASIC reads the count, detection is faster. Figure 16b shows reaction time for different ηs. Overall, the impact of η is low as the majority of the reaction time is due to measuring all of the ports and ensuring isolation. We contrast the above results to typical control plane failure detectors that require 10s of ms to detect failures and an additional few ms to route around them [28]. We also contrast the results to an idealized detection algorithm [15], which would be limited by sampling accuracy rather than detection latency. For example, η=20% and T_s=1 µs implies a minimum reaction time of

15 µs. Waiting for consecutive threshold violations would increase this time. The slightly lower latency comes at the cost of the benefits of control plane route recomputation.

8.3.3 Use Case #3: Hash Polarization Mitigation

Our third example is inspired by conversations with production network operators who noted the need to tune ECMP hashing functions for optimal load balancing given a particular packet header distribution. P4R can be used to reconfigure the inputs to a hash function at runtime to shift the function over time.

It can accomplish this by replacing the '5-tuple' input into the ECMP hash function with five malleable fields, each of which can become a reference to alternative fields in the packet's headers. To constrain the number of instantiated field_lists, we apply the optimization from the end of Section 4.1. The reaction function then takes a register array of per-egress packet counters and computes the Median Absolute Deviation (MAD) of the port utilizations. When the MAD differs for a sufficient amount of time, it shifts the inputs to find a better hash configuration for the current workload.

Compared to a control plane implementation that polls egress counters, Mantis provides isolation guarantees, which have been shown to be critical when evaluating ECMP balance [53, 57]. Compared to a data plane implementation that does the comparison in-band, the algorithm requires two operations that are difficult on today's switches. The first is the need to propagate egress counters (where packet counts can be observed) to the ingress (where routing decisions are possible), which often requires a recirculation. The second is the MAD computation, which traditionally requires computing the data's median, then the median difference from that value. A streaming algorithm suitable for use in networks would likely require us to use estimates of the median over only prior data and assume that value does not change significantly over time [38].

8.3.4 Use Case #4: Reinforcement Learning

Finally, we note that Mantis's reaction abstraction is a good fit for feedback loops like those of Reinforcement Learning (RL). More formally, in each iteration i, the Mantis agent measures data plane state, s_i, and reacts in some way, a_i. The state then transitions to s_{i+1} resulting in a scalar reward r_i. During execution, Mantis will make observations of the form $e_i = (s_i, a_i, r_i, s_{i+1})$ and adjust to maximize the expected cumulative reward.

Many tasks can fit into the above framework, but as an example, we consider the task of tuning the DCTCP ECN threshold heuristic [3] to optimize the sum of the utilization of the switch with the inverse of queue length. We do this via off-policy Q-learning [46]. Specifically, we cast the ECN marking threshold as a malleable value, configurable from the control plane via reactions a_i, and poll queue depth and a counter register from the egress pipeline as the observed state s_i. At each step, Mantis uses an ϵ-greedy policy to either exploit or explore the space; updates of the state-value function follow the TD control algorithm in [46].

Although others have proposed in-network RL previously, these solutions have tended to rely on custom accelerators [26]. RL is difficult in existing switches both because of the need for a feedback loop and the extremely limited computational ability of switch ALUs. Instead, Mantis-based RL can leverage the CPU and can easily extend to arbitrary models, including neural networks.

9 RELATED WORK

Over the years, a sequence of influential work [6, 31, 47] has provided network operators with an increasing amount of control at an increasingly fine granularity. A subset has also looked at control plane latency, though usually in the context of SDNs [8, 45].

Workarounds for data plane limitations. We are not the first to observe the limitations of today's programmable switches [4], and a slew of recent work has proposed data plane approximations/workarounds for specific building blocks [39, 40, 44]. While both innovative and effective, it is not clear that every protocol can be adapted to a pure P4-model, nor is it clear that every operator will have the time/expertise to develop an adaptation. Our work takes a different approach, allowing users to use arbitrary C as long as the application fits in the 'reaction' paradigm.

A subclass of the above workarounds involves the control plane in the workaround for precisely the reasons Mantis does [41, 52, 53, 56]. Mantis is a generalization of these proposals and one that presents a convenient (and potentially finer-grained) abstraction.

Alternative hardware architectures. Similar in spirit is work that has proposed alternative hardware solutions to the problem of data plane expressivity. Some of these propose and use modifications to RMT-style switches [15, 16, 22, 33, 42]; others propose the use of distinct hardware architectures such as FPGAs [9, 18, 30]. Unfortunately, given the current trends of network bandwidth versus compute power, it is unlikely that future switches and routers will be both line-rate and Turing-complete. In contrast, Mantis strives to provide a high degree of expressiveness on today's RMT switches.

Update and measurement isolation. Ensuring consistency and isolation of network updates is a classic problem in traditional and SDN networks, and many solutions have been proposed in those domains [10, 19, 35]. Some of these have also used a two-phase protocol [1, 23, 35]; however, as mentioned in Section 5.1.2 our focus on frequent, repeated updates of the data plane distinguish our design, implementation, and optimizations. Related work in the data plane has instead tended to focus on cross-pipeline consistency [53] and intra-pipeline atomicity [43].

Data plane virtualization. Finally, prior work has also observed the utility of match tables for runtime modifications [13, 54]. As mentioned in Section 3, however, this comes at a high cost. Instead, we target reactions in which most of the data plane is fixed, with only a few parameters dependent on current network conditions.

10 CONCLUSION

In this paper, we describe Mantis, a framework that reformulates common network tasks as reactions to current network conditions. We show that the Mantis compiler and control plane architecture enable fine-grained RTT-level reaction loops, while the P4R language simplifies the process of designing and implementing reactions.

ACKNOWLEDGMENTS

We gratefully acknowledge Vladimir Gurevich, Changhoon Kim, our shepherd Manya Ghobadi, and the anonymous SIGCOMM reviewers for all of their help and thoughtful comments. This work was supported in part by Facebook, VMWare, NSF grant CNS-1845749, and DARPA Contract No. HR001117C0047.

REFERENCES

[1] Richard Alimi, Ye Wang, and Y. Richard Yang. 2008. Shadow Configuration as a Network Management Primitive. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM '08)*. Association for Computing Machinery, New York, NY, USA, 111–122.

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 503–514. https://doi.org/10.1145/2619239.2626316

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). *SIGCOMM Comput. Commun. Rev.* 40, 4 (Aug. 2010), 63–74.

[4] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. 313–323.

[5] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10)*. Association for Computing Machinery, New York, NY, USA, 267–280.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and et al. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95.

[7] Caida. 2019. The CAIDA UCSD Statistical information for the CAIDA Anonymized Internet Traces. https://www.caida.org/data/passive/passive_trace_statistics.xml. (2019).

[8] Huan Chen and Theophilus Benson. 2017. Hermes: Providing Tight Control over High-Performance SDN Switches. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '17)*. Association for Computing Machinery, New York, NY, USA, 283–295.

[9] Daniel Firestone, Andrew Putnam, Hari Angepat, Derek Chiou, Adrian Caulfield, Eric Chung, Matt Humphrey, Kalin Ovtcharov, Jitu Padhye, Doug Burger, Dave Maltz, Albert Greenberg, Sambhrama Mundkur, Alireza Dabagh, Mike Andrewartha, Vivek Bhanu, Harish Kumar Chandrappa, Somesh Chaturmohta, Jack Lavier, Norman Lam, Fengfen Liu, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Kushagra Vaid, and David A. Maltz. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[10] K. Foerster, S. Schmid, and S. Vissicchio. 2019. Survey of Consistent Software-Defined Network Updates. *IEEE Communications Surveys Tutorials* 21, 2 (Secondquarter 2019), 1435–1461. https://doi.org/10.1109/COMST.2018.2876749

[11] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. 2017. DRILL: Micro Load Balancing for Low-Latency Data Center Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 225–238.

[12] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-Driven Streaming Network Telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 357–371. https://doi.org/10.1145/3230543.3230555

[13] David Hancock and Jacobus van der Merwe. 2016. HyPer4: Using P4 to Virtualize the Programmable Data Plane. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*. Association for Computing Machinery, New York, NY, USA, 35–49. https://doi.org/10.1145/2999572.2999607

[14] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 29–42.

[15] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. 2020. Contra: A Programmable System for Performance-aware Routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 701–721. https://www.usenix.org/conference/nsdi20/presentation/hsu

[16] Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. 2019. Event-Driven Packet Processing. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets '19)*. Association for Computing Machinery, New York, NY, USA, 133–140.

[17] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, and et al. 2013. B4: Experience with a Globally-Deployed Software Defined Wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 3–14.

[18] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. 2014. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 3–14.

[19] John P. John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. 2008. Consensus Routing: The Internet as a Distributed System. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, USA, 351–364.

[20] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. 2020. Programmable In-Network Security for Context-aware BYOD Policies. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Boston, MA.

[21] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. 2014. Design Guidelines for Domain Specific Languages. *CoRR* abs/1409.2378 (2014). arXiv:1409.2378

[22] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proceedings of the Symposium on SDN Research (SOSR '16)*. Association for Computing Machinery, New York, NY, USA, Article 10, 12 pages.

[23] Naga Praveen Katta, Jennifer Rexford, and David Walker. 2013. Incremental Consistent Updates. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*. Association for Computing Machinery, New York, NY, USA, 49–54.

[24] Junaid Khalid and Aditya Akella. 2019. Correctness and Performance for Stateful Chained Network Functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 501–516. https://www.usenix.org/conference/nsdi19/presentation/khalid

[25] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 1–14.

[26] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating Distributed Reinforcement Learning with In-Switch Computing. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 279–291.

[27] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. 2013. Ensuring Connectivity via Data Plane Mechanisms. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, Lombard, IL, 113–126.

[28] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. 2013. F10: A Fault-Tolerant Engineered Network. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, USA, 399–412.

[29] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 101–114. https://doi.org/10.1145/2934872.2934906

[30] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. 2007. NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education (MSE '07)*. IEEE Computer Society, USA, 160–161.

[31] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (March 2008), 69–74.

[32] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 221–235.

[33] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 85–98.

[34] P. Phaal, S. Panchen, and N. McKee. 2001. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176 (Informational). (2001).

[35] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer*

Communication (SIGCOMM '12). Association for Computing Machinery, New York, NY, USA, 323–334.

[36] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. 2011. Consistent updates for software-defined networks: Change you can believe in!. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. 1–6.

[37] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 123–137.

[38] shabbychef (https://stats.stackexchange.com/users/795/shabbychef). [n. d.]. Online algorithm for mean absolute deviation and large data set. Cross Validated. ([n. d.]). https://stats.stackexchange.com/q/3378 URL:https://stats.stackexchange.com/q/3378 (version: 2010-10-07).

[39] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Changhoon Kim, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. 2017. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, USA, 67–82.

[40] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 1–16.

[41] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable Calendar Queues for High-speed Packet Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 685–699.

[42] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 15–28.

[43] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 15–28.

[44] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. Association for Computing Machinery, New York, NY, USA, 164–176.

[45] John Sonchack, Jonathan M. Smith, Adam J. Aviv, and Eric Keller. 2016. Enabling Practical Software-defined Networking Security Applications with OFX. In *23rd Annual Network and Distributed System Security Symposium (NDSS '16)*. Internet Society.

[46] Richard S Sutton et al. [n. d.]. *Introduction to reinforcement learning*. Vol. 135.

[47] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. 1997. A survey of active network research. *IEEE Communications Magazine* 35, 1 (Jan 1997), 80–86. https://doi.org/10.1109/35.568214

[48] The P4 Language Consortium. 2018. The P4 Language Specification. (2018). https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf

[49] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, USA, 407–420.

[50] WG802.1. 2011. IEEE 802.1Qbb: Priority-based flow control. (2011). https://1.ieee802.org/dcb/802-1qbb/

[51] Dingming Wu, Ang Chen, T. S. Eugene Ng, Guohui Wang, and Haiyong Wang. 2019. Accelerated Service Chaining on a Single Switch ASIC. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets '19)*. Association for Computing Machinery, New York, NY, USA, 141–149.

[52] Jiarong Xing, Wenqing Wu, and Ang Chen. 2019. Architecting Programmable Data Plane Defenses into the Network with FastFlex. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets '19)*. Association for Computing Machinery, New York, NY, USA, 161–169.

[53] Nofel Yaseen, John Sonchack, and Vincent Liu. 2018. Synchronized Network Snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 402–416.

[54] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. 2017. HyperV: A high performance hypervisor for virtualization of the programmable data plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–9.

[55] Irene Y. Zhang. 2017. Operation Ordering in Systems. (2017). https://irenezhang.net/research/consistency.html

[56] Menghao Zhang, Guan-Yu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. 2020. Poseidon: Mitigating Volumetric DDoS Attacks with Programmable Switches.

[57] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-resolution Measurement of Data Center Microbursts. In *Proceedings of the 2017 Internet Measurement Conference (IMC '17)*. ACM, New York, NY, USA, 78–85.