Flightplan: Dataplane Disaggregation and Placement for P4 Programs

Nik Sultana John Sonchack Hans Giesen Isaac Pedisich Zhaoyang Han
Nishanth Shyamkumar Shivani Burad André DeHon Boon Thau Loo
University of Pennsylvania

Abstract

Today's dataplane programming approach maps a whole P4 program to a single dataplane target, limiting a P4 program's performance and functionality to what a single target can offer. Disaggregating a single P4 program into subprograms that execute across different dataplanes can improve performance, utilization and cost. But doing this manually is tedious, error-prone and must be repeated as topologies or hardware resources change.

We propose Flightplan: a target-agnostic, programming toolchain that helps with splitting a P4 program into a set of cooperating P4 programs and maps them to run as a distributed system formed of several, possibly heterogeneous, dataplanes. Flightplan can exploit features offered by different hardware targets and assists with configuring, testing, and handing-over between dataplanes executing the distributed dataplane program.

We evaluate Flightplan on a suite of in-network functions and measure the effects of P4 program splitting in testbed experiments involving programmable switches, FPGAs, and x86 servers. We show that Flightplan can rapidly navigate a complex space of splits and placements to optimize bandwidth, energy consumption, device heterogeneity and latency while preserving the P4 program's behavior.

1 Introduction

Different kinds of hardware can be leveraged to make networks programmable, including CPU-based servers running "software-ized" Network Functions (NFs), NPUs, FPGAs and programmable ASICs. Although these hardware targets have complementary strengths when it comes to performance, flexibility, and power utilization, it is difficult to combine their strengths. Many NFV frameworks are CPU-centric, and chaining services across diverse hardware usually treat the hardware's capabilities as a black box. This is partly because toolchains for NPUs, FPGAs, and ASICs are alien to most software developers.

Code, documentation, tests and data can be downloaded from: https://flightplan.cis.upenn.edu

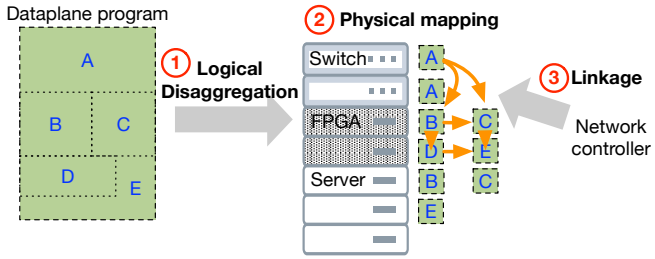


Figure 1: In this illustration, a program is split into 5 logical parts, A-E. ① A program is annotated with logical delimiters, manually or automatically. Flightplan splits the program into complementary parts using these annotations and provides coordination and linkage code between these parts, which the Flightplan control program configures at runtime. ② Each part of the original program is mapped to a physical device. In this illustration, A is mapped to Top-of-Rack (ToR) switches, B, C D, and E to network-attached FPGAs, and redundant instances of B, C, and E are mapped to execute on server CPUs. ③ The Flightplan control program can alter the program's linkage at runtime, to use different hardware targets, mitigate faults, or balance load.

While P4 is emerging as a common language for programming dataplanes across programmable ASICs, NPUs, FPGAs, and CPUs, P4 programs are limited to what can be run on a single target because they are programmed in an approach that maps a whole program to a single dataplane. This approach limits a dataplane program to what can be computed using a single target's resources and capabilities.

Through testbed experiments we measured how splitting a single P4 program into subprograms that execute across different dataplanes can improve performance, utilization and cost (§2, §7.2.3).

In principle, one could write a set of P4 programs that execute jointly across different dataplanes, combining their strengths. Further, P4 could be used as a convenient syntax for both NF authoring and for NF composition across different types of hardware. This set of P4 programs would be written

to follow this pattern:

- If the dataplane program exceeds the resources provided by a single hardware target, then part of the program could be disaggregated to run on additional targets.
- If the dataplane program exceeds the capabilities provided by a *class* of targets, it could be disaggregated to run on two or more heterogeneous targets (e.g., ASIC and FPGA).
- Redundant spares can be provisioned for quick fail-over, leveraging the best available hardware for a given dataplane program.

However, writing distributed P4 programs manually is tedious and error prone. Current dataplane programming approaches lack abstractions for inter-program communication and mechanisms such as RPC [5] across different in-network functions. This complicates the implementation and composition of sophisticated in-network services because dataplane programmers are burdened with having to write and manage such coordination explicitly. It is also laborious to re-distribute the P4 program when new equipment or NFs become available.

We propose and evaluate Flightplan, a target-agnostic, P4-based programming toolchain that disaggregates P4 dataplane programs into a set of dataplane subprograms and runs them as a distributed system formed of several, possibly heterogeneous, dataplanes. It uses hardware performance and resource profiles to plan the allocation of subprograms onto dataplanes in the network on which to execute. The composite behavior of the resulting distributed dataplane program is the same as the original program; coordination and synchronization code is provided by the Flightplan runtime.

We reduce the P4 program, hardware performance profiles, and network topology constraints to a common rule-based formalism that Flightplan uses to map parts of the original program to dataplanes in the network. Through evaluation we show that it rapidly navigates a complex space of configurations which are then ranked according to the sought optimization criteria, and show that several disaggregated programs can be run simultaneously in the same network.

Figure 1 sketches our approach. It uses unmodified, commodity hardware and does not require changing the P4 language. A key enabler of this approach is that P4 enjoys toolchain support to target diverse hardware: ASIC [3], FPGA [23, 37, 38], NPU [24], and CPU [25]. These diverse targets' toolchains are based on P4's reference compiler [26], which we extend in our prototype. Unlike traditional NF service chaining [28], Flightplan works over diverse hardware.

We summarize our key contributions as follows:

- **Dataplane program disaggregation.** (§2) We propose the concept of dataplane program disaggregation and provide a motivating use-case. We show how disaggregation enables better utilization of existing resources.

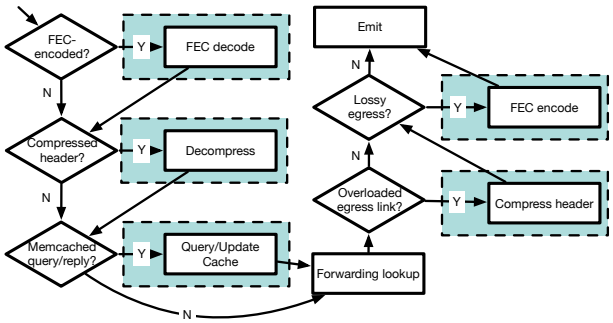


Figure 2: Our example dataplane program, `Crosspod.p4`, shown as a flowchart for compactness. The colored dashed rectangles surround functions that one might need to offload to other dataplanes in the network to free up resources on the switch, or because they exceed the computational abilities of the switch.

- **Automation for disaggregation.** (§3) We present a novel approach that partly automates dataplane disaggregation and does not require changes to the P4 language or to hardware targets and their toolchains. We extend the open-source P4 compiler to implement a program analyzer (§4) that discovers resource-use and dependencies that must be preserved once the program is split.
- **Flightplan runtime support.** (§5) We describe the requirement of in-dataplane and out-of-dataplane runtime support for disaggregated programs and how the runtime influences the process through which programs are disaggregated. We explore the design space by implementing 3 runtimes for Flightplan, offering different trade-offs between features and overhead.
- **Flightplan planner.** (§6) We implemented a prototype tool that generates configuration plans for disaggregated programs.
- **Evaluation.** (§7) We present a detailed evaluation of different aspects of this work, including testbed experiments involving heterogeneous hardware and microbenchmarks to measure resource transfer and disaggregation overhead.

2 Motivating Example: Crosspod

`Crosspod.p4` is an 800-line P4 program, sketched in Fig. 2, that will provide a running use-case. After describing how `Crosspod.p4` works, we describe the process of disaggregating it into several subprograms to run on three different classes of hardware.

We wrote `Crosspod.p4` to improve network reliability and performance using caching, compression, and forward-error correction in a way that is transparent to applications and users. Figure 3 shows its execution. Figure 4 shows two examples of disaggregation for this program.

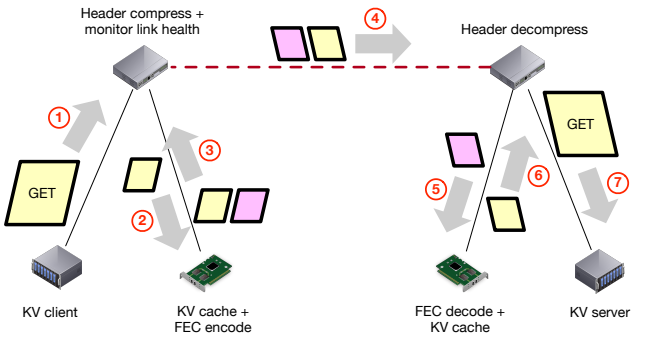


Figure 3: ① Key-Value (Memcached) client generates a GET request (yellow packet) which it puts onto the network for the KV server to respond to. ② A transparent, rack-level in-network KV cache is consulted. ③ In the event of a cache miss then the request is relayed onwards. The switch compresses its header, and upon detecting that the request shall cross a lossy link, it activates link-layer FEC which is computed in the network using reconfigurable hardware. ④ The KV packet and FEC-related redundancy (pink packet) cross the channel to the next switch. ⑤ Lost packets can be recovered during FEC decoding. ⑥ The packet's header is decompressed. A second inline KV can be consulted, to take pressure off the host-based KV server. ⑦ If all the caches were missed, then the request finally reaches the KV server, which sends its response back to the requesting client.

In-network functions. `Crosspod.p4` invokes a set of in-network functions to achieve its goal. Some of these functions are external to P4, and we implemented them to run on different types of hardware. In our example dataplane program, *reliability* is improved by (1) using forward-error correction (FEC) to mitigate faulty links; (2) application-specific caching to lessen congestion; and (3) header compression to lessen congestion. The *performance* of the network is improved by (1) reducing link utilization (through caching and header compression), (2) reducing latency (through caching closer to clients), and (3) reducing server utilization (through caching). In particular, caching is directed at Key-Value (KV) queries, a staple service in modern datacenter systems [10, 18, 21, 35].

Why Dataplane Disaggregation. In this example dataplane program, we combine in-network functions that cannot entirely be carried out within a single type of hardware for the following reasons: (1) **resources**: we cannot run the program entirely on a programmable switch ASIC because some of the functions (e.g., layer 2 FEC) exceed the computations that can be carried out in state-of-the-art devices, which typically do not include payload processing; (2) **performance**: we cannot run it entirely on an FPGA because this would severely constrain the throughput of this program. ASICs often support higher I/O bandwidth per chip and use less silicon for the standard dataplane switching operations, resulting in fewer or less expensive chips to handle the highest throughput data

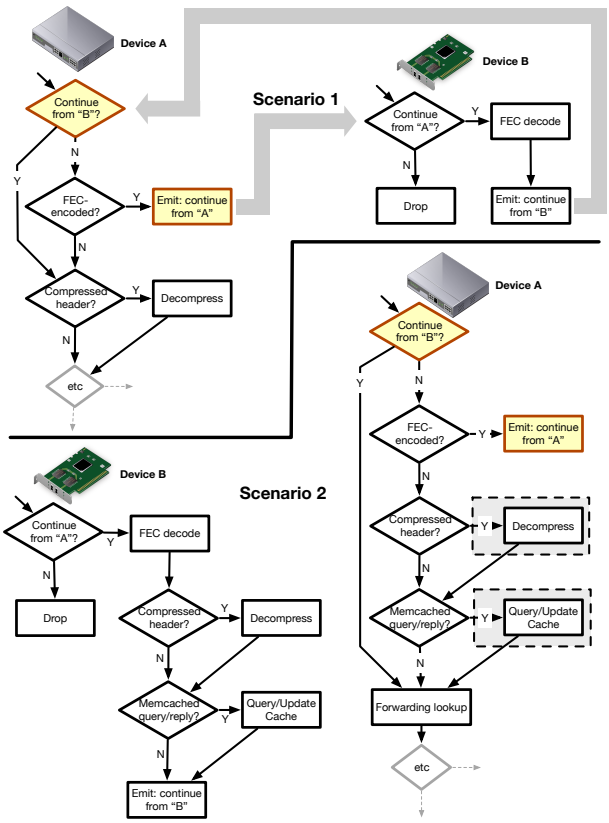


Figure 4: Two ways of splitting the program from Figure 2 between two devices: in **Scenario 1** a single function is offloaded from the switch (Device A) to an FPGA board (Device B) that immediately returns control back to the switch after executing the function; in **Scenario 2** a segment of the dataplane program is offloaded from the switch to the FPGA, which carries out several functions.

movement; (3) **expense**: even if we could place a program entirely on a single hardware dataplane, we do not want to use up resources unnecessarily and would prefer to move less-traversed code to a less expensive dataplane (e.g., an end-host). Conversely, if we have underutilized FPGAs, then we might prefer to use them rather than an end-host to save on power and cooling costs [14]. Last, (4) **availability**: we might want to failover quickly and autonomously from centralized monitoring and reconfiguration, by installing logic into the dataplane for self-management.

Why Automate Dataplane Disaggregation? Disaggregating a program involves i) deciding how to split it, ii) adapting the derivative subprograms to hand-over to one another, and iii) placing the subprograms on targets in the network.

Table 1 shows different characteristics of different hardware targets when executing the same function on the same workload. Automation spares the network operator from having to manually pick hardware combinations and track their

	Throughput (Gbit/s)			Power (W)
	Compression	FEC	KV Cache	
P.ASIC	9.07	-	-	110.5
FPGA	8.35	7.95	7.73	27.3
CPU	0.10	0.04	-	140.6

Table 1: Maximum Throughput and Average Power of network functions running on different hardware. Dashes indicate that we do not have implementations for network function to run on a particular target.

utilization.[1] In addition to power and performance, Flightplan can optimize for unit cost, utilization and latency.

The difficulty of this task is compounded by the likelihood that the choice of programs, topologies and target devices will change over time, requiring the whole process to be redone. If several programs are being disaggregated, then it becomes more challenging to optimize their placement jointly. Further, placement constraints and objectives may change over time, too: changing priorities over which links to protect with redundancy for example, or which traffic to compress, might require different function placements in the network.

This practical difficulty makes a strong case for automating dataplane disaggregation. But automation needs to be done carefully to avoid incurring the theoretical complexity of the automatic disaggregation problem. We estimate this to be exponential in the number of targets available on which to map subprograms and doubly exponential in the number of subprograms.[2] Flightplan uses heuristics to avoid this blow-up and our prototype also emits coarsening advice to opportunistically decrease the split granularity to better utilize the available hardware.

Deployment practicalities. The design and evaluation of Flightplan addresses the following practical considerations: (i) Multiple disaggregated programs can be used in the same network simultaneously: our evaluation in §7 describes how we ran 10 P4 programs in the same network, 6 of which were disaggregated, and of which 4 were different disaggregations of the same program. (ii) Upgrades can be done in a phased manner, as standard in deployment [34], by running the old and new versions of software simultaneously in the same network as described above. (iii) Debugging is done using standard techniques—inspecting packet traces, counters, etc.—complemented by using the Flightplan *control program* (§5) to conveniently query Flightplan-specific state of the disaggregated program across all the dataplanes on which parts of it are running. (iv) Failures can be detected and handled by Flightplan runtime support inserted into the dataplane itself or by remotely using the control program.

[1] More detailed information is provided in Table 3 in Appendix B.
[2] The full calculation is given in Appendix C.

3 Flightplan Overview

Flightplan produces a sequence of plans, consisting of a disaggregation of a dataplane program into multiple programs, and the allocation of these programs to dataplanes in the network. A plan targets the Flightplan *runtime* which provides the facilities to configure, start, and execute the disaggregated program. The role and design of Flightplan's runtime support is detailed in §5.

In this section, we outline all the inputs and outputs of Flightplan before going into more detail in the sections that follow. Figure 5 illustrates our workflow. ① We start with a P4 program and *segment* it. Segments are sequences of statements from the original program and provide the planner with the smallest granularity of program parts that it can then map to different targets. Figure 5 shows the program being decomposed into five segments as an example, labeled A-E. The `then` and `else` blocks of an `if` statement can go into separate segments, as hinted in the drawing in Figure 5 where segment A branches to B and C.

Programs are currently segmented manually, but this could be automated in the future. Segmentation consists of inserting additional lines in the program that are interpreted specially by our extension of the P4 compiler to delimit segment boundaries (§4.1.1). ② Our P4 compiler extension analyzes segments to generate a set of Prolog-like rules that expresses an *abstract program* completely automatically (§4.1.2). The abstract program consists of a DAG of segments, with P4 code replaced by the abstracted resources it depends on. Abstracted resources, such as tables and external functions, might not be available on all hardware or might have different capacity, power, throughput, and latency characteristics when invoked on hardware.

③ The *abstract resource semantics* is an additional set of rules used to check whether a dataplane satisfies a segment's resource needs. We use these rules to encode the performance profile which was summarized in Table 1. The contents and generation of rules is explained further in §4.

④ The planner is provided with a description of the network, including topology and device information, expressed in the same formal language used for our abstract semantics, and ⑤ objectives to optimize, consisting of variables changed by the abstract resource rules—such as Rate in Rule 1. ⑥ If the constraints can be satisfied for the given topology, abstract resource rules and segmented program, then the planner will lazily and exhaustively generate all plans to optimize objectives. A *plan* consists of three components. The *allocation model* describes how the abstract state—such as packet size and latency—is modelled to change as the program executes across dataplanes. This is used to understand the allocation that the planner has found. The *annotated program* consists of the original program with possibly coarsened segments, reflecting how the segments are going to be split into subprograms. In this example, segments B and C are united into F

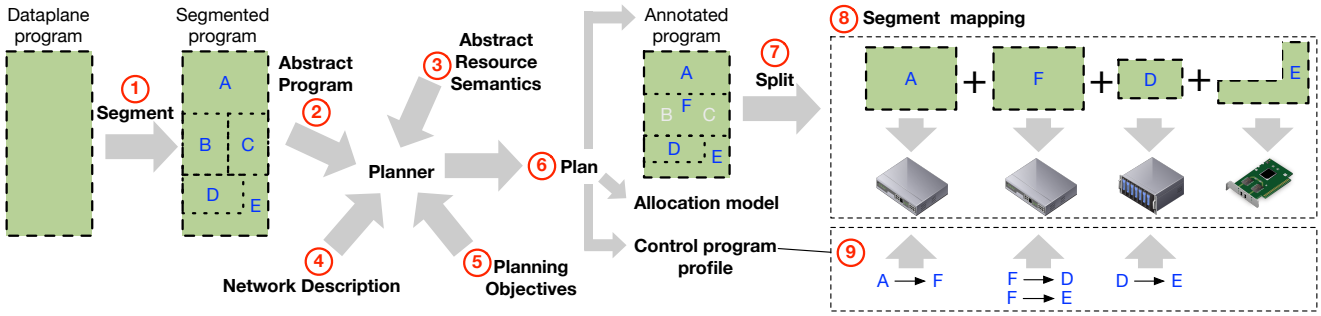


Figure 5: Flightplan's workflow, described in §3.

$$\frac{\begin{array}{c}\text{CPU} \quad \text{Rate} < 2 \times 10^8 \\ \text{PacketSize} > 1000\end{array}}{\text{header\_compress}} \left[\begin{array}{l}\text{Lat.} \mapsto \text{Lat.} + 7.4 \times 10^{-3} \\ \text{Rate} \mapsto \text{Rate} \times \frac{189.9}{194.75} \\ \text{once Power} \mapsto \text{Power} + 150\,\text{W} \\ \text{once Cost} \mapsto \text{Cost} + 5\end{array}\right]$$

Rule 1: This rule states that we can execute Header Compression (HC) on packets *if* we are running on a CPU, and the throughput and packet size are within given bounds. We form different rules to describe the same function operating under different bounds—e.g., different Rate or PacketSize—and on different hardware targets, such as our FPGA implementation. If this rule can be used then the *effect* of executing HC in this instance is shown in the [...]-enclosed finite function on user-defined $\mathbb{R}$-valued variables. In addition to adding latency (Lat.), the function reduces Rate because of its compression of network traffic. 'once' indicates that a function is only executed once whenever using a specific target—in this case using an x86 server raises power estimate by 150 W regardless of how many segments are allocated to that particular target. 'Cost' is a normalized unit cost we use for different types of devices on our network. The [...]-enclosed function is allowed to mutate the planner's representation of the program's state, modelling the effect of invoking the resource.

for mapping to the same dataplane. Finally, the *control program profile* will be used by Flightplan's control program to configure the distributed program's runtime, start it, and query its state. The profile contains port, state information, and segment information, and is specific to a disaggregated dataplane program. ⑦ The annotated program is split into subprograms. This process involves augmenting each segment with a copy of the Flightplan runtime to configure, test, and hand-over between these programs. Some dataplane functions may be external to the P4 program, such as the FEC from the previous section. Flightplan treats such functions as black boxes, and it does not generate non-P4 code for specific targets, such as FPGAs or CPUs. ⑧ The program segments are compiled using the target-specific toolchain provided by the target's vendor. ⑨ The control program configures the Flightplan runtime of each target on which a segment is allocated by *linking* the segments together: i.e., indicating on which port to hand-over to its peer segments. Once the system's configuration is complete, the control program can start its execution.

Listing 1: Snippet from `Crosspod.p4` (§2) showing an example segmentation. Highlights show segmentation annotations in orange and resource-related syntax in green.

```
bit<1> compressed_link = 0;
bit<1> run_fec_egress = 0;
...
flyto(Compress);
// If heading out on a multiplexed link, then header compress.
egress_compression.apply(meta.egress_spec, compressed_link);
if (compressed_link == 1) {
    header_compress(forward);
    if (forward == 0) {
        drop();
        return;
    }
}
flyto(FEC_Encode);
check_run_FEC_egress.apply();
// If heading out on a lossy link, then FEC encode.
if (run_fec_egress == 1) {
    ...
    classification.apply(hdr, proto_and_port); // Sets hdr.fec.isValid()
    if (hdr.fec.isValid()) {
        encoder_params.apply(hdr.fec.traffic_class, k, h);
        update_fec_state(hdr.fec.traffic_class, k, h,
                         hdr.fec.block_index, hdr.fec.packet_index);
        hdr.fec.orig_ethertype = hdr.eth.type;
        FEC_ENCODE(hdr.fec, k, h);
        ...
```

4 Rules in Flightplan

Rules of different kinds play a central role in Flightplan. These rules are combined to describe the abstract semantics of a P4 program and how the resources it needs to use are satisfied by hardware targets in a network.

Steps ②, ③ and ④ in Figure 5 involve producing rules for the planner to use. This section describes the content of these rules and how they are generated.

An important design feature in Flightplan is that rules do not have to be encoded literally by users. As explained in this section, rules are either created automatically from P4 programs by the Flightplan analyzer, or are generated from a table that describes the network and profiles of devices in the network. This input is then automatically converted into rules based on Definite Horn clauses [9] that rely on a simple propositional language that is explained further in §I.

4.1 Abstract Program

Our P4 program analyzer turns a P4 program into an abstract form consisting of a set of Prolog-like rules. To better describe the generation of rules for step ② we first elaborate on step ①.

4.1.1 Program Segmentation

Step ① involves adding demarcating statements to P4 code. These statements consist of calls to the special function 'flyto()', passing it a unique name to be used for the new segment. A segment extends until the next flyto() or the control block's end, whichever is reached first. flyto() statements have no effect in P4, they are interpreted by our P4 compiler extension which we refer to as the *Flightplan analyzer*. These statements provide syntactic markers that define the sub-program granularity for the rest of the planning process. Enclosing the whole program in a *single* segment is a degenerate segmentation that will require the planner to place the program *entirely* on a single dataplane for execution while satisfying all other constraints.

The snippet in Listing 1 shows three segments beginning at lines 4, 14, and implicitly at line 1. The first segment is named FlightStart by default.

Segmentation is done manually by the programmer or automatically by a tool. In our prototype the programmer manually segments the code, and subsequently the planner will merge contiguous segments if they are to be placed on the same dataplane.

Once a segmentation has been made, the Flightplan analyzer discovers data-dependencies through static analysis of P4 code and determines whether a segment break is allowed for the intended runtime. Flightplan runtimes will be detailed in a later section.

4.1.2 Program Abstraction

For a given segmentation we next generate rules in step ②. We devised a framework for *abstract* semantics for P4 that is focused on resource dependence. Each segment is reduced to the resources it needs to execute, and everything else is abstracted away. In Listing 1 such resource-related syntax is highlighted in green.

The Flightplan analyzer is a P4 compiler extension that carries out static analysis of P4 code to gather which resources are relied upon by each segment. Resources include invocation of external functions and table lookups. The analyzer then automatically generates a Prolog-like [36] set of rules for that segment. We call the collection of segments' rules the *abstract program* (§4.1) derived from the original P4 program. The abstract program is emitted into a JSON file that the analyzer parses; users do not need to write, inspect, or change the contents of this file.

One rule is generated for each segment and describes the resources used along each path through that segment.

Rule 2 describes the third segment from Listing 1 (Lines 4-13). In this case there are three paths through the segment: lines (5-7,13), (5-9,12,13), (5-13), and Rule 2 is showing the path whose requirements subsume those of other paths. The analyzer emits the resource requirements of each path, and

$$\frac{\text{egress\_compression} \quad \text{header\_compress} \quad \text{drop}}{\text{Compress}}\ [\text{Id}]$$

Rule 2: Program segments are abstracted into rules showing resource dependence. This rule states that segment Compress can be mapped if the target dataplane can provide implementations of egress_compression, header_compress and drop: these are satisfied by means of other rules that are provided at input. We saw a rule for header_compress in Rule 1. Id is the identify function: using this rule does not mutate abstract state.

$$\frac{\begin{array}{c}\text{CPU} \quad \text{Rate} < 8 \times 10^7 \\ \text{PacketSize} > 1050\end{array}}{\text{fec\_decode}} \left[\begin{array}{l}\text{Lat.} \mapsto \text{Lat.} + 0.09 \times 10^{-3} \\ \text{Rate} \mapsto \text{Rate} \times \frac{64.47}{77.36} \\ \text{once Power} \mapsto \text{Power} + 150\,\text{W} \\ \text{once Cost} \mapsto \text{Cost} + 5\end{array}\right]$$

Rule 3: Abstract resource rule for running fec_decode on a CPU.

the planner (§6) checks these are satisfied before allocating that segment to a prospective execution target.

4.2 Abstract Resource Semantics

In step ③ we supply the semantics of each program-used resource, such as calls to external functions, in terms of the measurable costs incurred for a program to use that resource on a specific hardware target. Such costs include latency, throughput, power, and the cost of the hardware.

In our prototype, the user encodes this information as a table of CSV entries, and can reuse this information across all invocations of Flightplan. A script then turns this table into a JSON encoding of the rules that are used by the planner. The measurements in these entries are derived empirically using the workflow described in §J. This involves carrying out profiling experiments that measure the characteristics of using resources on different hardware targets.

Rule 3 shows the characteristics of applying our FEC decoding function on a CPU. Rule 4 in §I shows a rule for applying Header Compression on an FPGA. Compared with Rule 1 it supports much higher throughput over smaller packets. Our table can be refined to include more details about the specific CPU and FPGA parts that were used, and capture other information about the target or the workload, without changing our general approach.

4.3 Network Representation

In step ④, the last set of rules states facts about the topology, the devices it comprises, and their ports. For example, that the proposition "CPU" holds on a specific network element, or the bandwidth limit of a specific port.

For a given port π, we gather facts in a set called π_{Provides}. We also associate constraints with π that must be satisfied for π to be crossed. We call this set π_{Requires}. [3]

[3] Examples of π properties are given in §I.

In our prototype we use JSON to encode the network's topology and the capabilities of each device and port. We reuse this information across all our invocations of Flightplan, and it only need to be changed when the network hardware or topology change.

5 Flightplan Runtime Support

A disaggregated program cannot function without runtime support. At the very least, the runtime provides the gluing code to *link* different parts of the disaggregated programs together for the computation to flow through them as it would in the original program.

The choice of runtime needs to be made first since it influences the rest of the process. This choice is communicated to the analyzer and splitter (§6.4). In Flightplan, the following information is in scope for the runtime to manage: (1) runtime metadata, such as (1a) which part of the program needs to be executed next, (1b) values of live variables to be preserved across dataplane hand-overs, (1c) state related to in-dataplane failure detection and handling, and (2) switch metadata—such as ingress and egress ports, since these might be read or written during the program's execution, including table lookups and actions.

There are different choices of runtime features and ways of implementing them. Our different runtime implementations show how the core idea in Flightplan—that of programmable-dataplane disaggregation—spans a design space and not a single implementation.

5.1 Runtime Design

Runtime support for Flightplan consists of two components, the first running outside the dataplane and the second running inside it. The first is a control program that configures the latter and queries its state. Configuration information is stored in register values and table entries used by the second component to support the execution of each segment of the distributed P4 program. In our prototype, the control program—called `fpctl`—emits dozens of commands to the P4 controller, which in turn interacts with the dataplane. The inputs to `fpctl` consist of the network topology, the control program profile, and a command (such as `configure` or `start`) and its parameters.

The second component runs inside the dataplane and must be written in P4. It is combined with each constituent of a distributed P4 program. In Fig. 1, each constituent A-E would be instantiating the same runtime, albeit in possibly heterogeneous dataplanes.

5.2 Runtime Diversity

We developed three runtimes, each representing a different point in a design space. The three are called 'Full' (§5.3), 'Headerless (IPv4)', and 'Headerless'. We use `fpctl` to interact with all of them. All our implementations use standard features of $P4_{16}$ and target the simple_switch BMv2 target [25], whose features intersect with most P4 devices.

The Full runtime uses a special Flightplan header, but the other two do not. Our IPv4 headerless approach steals bits from the IPv4 Fragment Offset header for Flightplan state. However, we found that this was not always usable since not all of our network traffic has IP headers—specifically the FEC parity frames lack these. This spurred us to develop a completely headerless approach, described in §5.4.

Recalling the example from Listing 1, the Full runtime allows us to preserve the value of `meta.egress_spec` by serializing it into the Flightplan header during hand-over. We can alternatively use the Headerless runtime since, while preserving switch metadata such as `meta.egress_spec` is generally not possible in our Headerless approach, we encode this information in the wiring and table entries. This limits the behavior of the original program, trading off flexibility for less overhead, since the Headerless approach essentially creates a circuit through dataplanes.

For a further example of how the Full runtime allows more flexibility for segmentation, we could add a flyto() between lines 6 and 7 in Listing 1. The Full runtime allows us to hand-over the value of `compressed_link` from Compress to the new segment, but the Headerless runtime would not be able to support this segmentation.

5.3 Full Runtime

The Full runtime provides the most flexibility and resilience among our runtimes. It uses a custom header to accommodate all of the features (1a-c,2) listed earlier in §5. The header's definition is provided in §D.

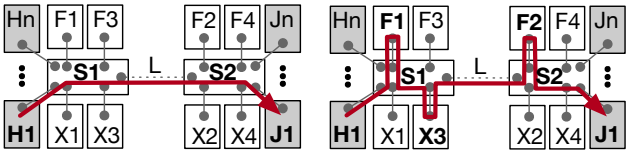
The header consists of two kinds of fields: scratch space in which metadata and program variables are serialized and state fields for encoding status flags and values, such as which segment from the original program is to be executed next.

Part of the distributed program might be unreachable because of network or node failures, thus compromising the overall program's execution. This runtime implements a feedback loop between connected dataplanes to push fault detection and handling into the dataplane. The details are in §E.1.

`fpctl` can be used to query the runtime's state, for instance to find out if it is close to meeting a fail-over threshold, or if it has failed-over. It can also overwrite this state and force a fail-over remotely.

5.4 Headerless Runtime

This approach does not encode any state about the ongoing computation across dataplanes. Consequently this runtime provides less flexibility than the 'Full' approach: metadata is not preserved, it does not support in-dataplane failure detection and handling, and values of live variables are not preserved—thus segments must be coarser to avoid needing to hand-over the variables' values. This runtime is described further in §E.2.



(a) Shortest path from H_1 to J_1. (b) A detour path from H_1 to J_1.

Figure 6: Abstract network used for running example in §6.

To compensate for the lack of header, we rely on the ingress and egress port information. An interesting consequence of the Headerless approach's feature-paucity is that it can be made to work with older SDN switches that are not P4 programmable. To enable this, `fpctl` generates flow-table configurations from the control program profile. We report on the use of non-P4 SDN switches in our testbed evaluation.

6 Flightplanner

Flightplan combines graph-based and formal methods to find execution plans for disaggregated programs over dataplanes in the network.

Execution of the disaggregated program occurs along paths in the network, and therefore the planner needs to be told about the network's topology and hardware capabilities. Fig. 6 resembles our physical testbed and is used in this section to help explain an execution plan. In this network, S_i are switches, F_i are network-connected FPGA boards, X_i are CPU-based network elements that can run P4 programs, and H_i and J_i are servers to which P4 programs cannot be offloaded.

Flightplan explores resources around switches to carry out detours in the forwarding along dataplanes onto which some of the computation can be offloaded as shown in Fig. 6b. In this example S_1 offloads computations to F_1 and X_3, and S_2 offloads to F_2.

6.1 Abstract Program State

The planner maintains a small amount of state as it explores plans. This state consists of values for a special set of variables V that appear in rules, such as PacketSize and Rate. Using these variables we form Bound expressions such as PacketSize $>$ 1000 from Rule 1.[4]

The *abstract program state* θ consists of a total map $V \to \mathbb{R}$ that encodes the value of each V at one instant during the program's execution. The values of V can be transformed by rules in their $[\ldots]$-function, while the values of propositions can only be derived through proof, as will be explained shortly. During the planner's execution, all Bound expressions are ground, making the evaluation process straightforward.

6.2 Proof-based Segment Allocation

Producing a plan involves two kinds of inference: (i) deciding whether a given dataplane can execute a given segment in a way that satisfies all related constraints; and (ii) finding a plan—a sequence of dataplanes over which all of a program's segments are executed. This section describes how item (i) is done in Flightplan. The next section builds on this to describe (ii).

[4] Further details of our modeling language are in §I.

$$\dfrac{\dfrac{\theta, \pi_{\text{Prov.}}}{\text{CPU}}\ \text{R5} \quad \dfrac{\theta, \pi_{\text{Prov.}}}{\text{CPU} \quad \text{Rate} < 2\times10^8}\ \text{R1} \quad \dfrac{\theta, \pi_{\text{Prov.}}}{\text{PacketSize} > 1000}\ \text{R6}}{\text{egress\_compres.}} \quad \dfrac{\text{CPU}}{\text{header\_compress}} \quad \dfrac{\text{CPU}}{\text{drop}}}{\text{Compress}}\ \text{R2}$$

Proof 1: Flightplan may allocate segment Compress to π only if a proof can be derived using θ, program+resource+network rules and π_{Provides}.

For a given port π (§4.3), to decide whether a segment Seg can be mapped to π's dataplane, we need to do two things. First, ensure that all of π's constraints are satisfied. For example, the current transmission rate must not exceed the ports limit. Second, we use facts provided by π to ensure that Seg is *derivable* from the rules we have available. For example, all external functions called in Seg must have viable implementations once they cross that port.

This derivation involves building a Prolog-style formal proof. For example, if $\pi_{\text{Provides}} = \{\text{CPU}\}$ then Compress is derivable as shown in Proof 1 if θ satisfies the bound-related constraints of the rules used in this derivation.

In addition to obtaining assurance that a target can execute a segment, we use proofs to compute the transformations of abstract program state. This involves composing the $[\ldots]$-enclosed functions by doing a post-order traversal of the proof tree, then applying this function to the search state θ to obtain the new search state. For Proof 1 the state transformation γ is $\gamma = \gamma_{\text{R2}} \circ \gamma_{\text{R6}} \circ \gamma_{\text{R1}} \circ \gamma_{\text{R5}}$

6.3 Plan-finding

Given a DAG of abstract program segments (§4.1) we search for a succession of dataplanes that packets can be made to traverse such that all program segments can be executed over those packets. Since devices in the same class are identical, we factor the solution space by the different device classes to avoid returning quasi-duplicate solutions. The user can choose whether they want the best solution—by having the tool explore all possibilities. Alternatively, they are given the solution found using a simple greedy heuristic on optimization objectives—for example, by choosing the next dataplane that least increases latency.

The network operator needs to provide three additional pieces of information: (i) an initial abstract program state θ_0 (§6.1), (ii) the switch on which the disaggregation program's execution will be centered (e.g., S_1 in Fig. 6), and (iii) the set of devices to which the switch may offload to (e.g., $\{F_1, F_3, X_1, X_3\}$ in Fig. 6).

The planner carries out a breadth-first search while attempt-

ing to allocate segments as described in the previous section. At each hop it updates the abstract state using γ to compute an approximation of resource-usage across the network. This will be used to evaluate constraints in the rest of the potential plan.

When a plan is found, it is converted into the three outputs shown in Figure 5. First, contiguous segments that are to be mapped to the same target are unified into larger, coarser-grained segments. Second, the *allocation model* is produced by emitting the trace of abstract program states and the mapping from segment names to targets in the network. Finally, a profile is produced for use by `fpctl`—Flightplan's control program—to configure, start, and query the disaggregated P4 program. A profile consists of a mapping of generated P4 programs to a subset of the network. Flightplan uses the profile to configure a dataplane target through the target-specific control plane interface.

6.4 Program Splitting

Next we generate a separate, well-formed and self-contained P4 file for each segment. The program splitting phase performs three tasks: 1) extract the P4 code from each segment, forming subprograms; 2) analyze the subprograms to gather runtime-related context, such as variables whose values must be included in the hand-over between dataplanes; 3) inject runtime-dependent code for handing-over between subprograms. These are described further in §K. The Flightplan analyzer prototype also emits split programs satisfying points 1 and 2. The interfacing to the runtime can be automated in the future.

7 Evaluation

We evaluate Flightplan using virtual and physical networks to answer various questions about its features and the implications of disaggregation choices.

7.1 Scale, Overhead and Disaggregation

We use a virtualized network (§7.1.1) to test logical qualities of a Flightplan deployment, using P4 applications both that we wrote and from third-party sources.

To add realism to our experiments, we implemented a generator for complete configurations of fat-tree networks [1] to run on our setup, which is built on Mininet [19] and BMv2 [25]. We used $k = 4$ in this evaluation.

We implemented routing logic for this topology in a P4 program called `ALV.p4`. It provides a baseline P4 program that implements minimal functionality in the network—routing. We embedded it in four other P4 programs: (i) `Crosspod.p4` (§2), (ii) `firewall.p4`, (iii) `qos.p4` and (iv) `basic_tunnel.p4`. The latter three are third-party open-source programs from the P4 tutorial repository [27].

Initially our virtual network had `ALV.p4` executing on all switches. We then installed the other P4 programs on some

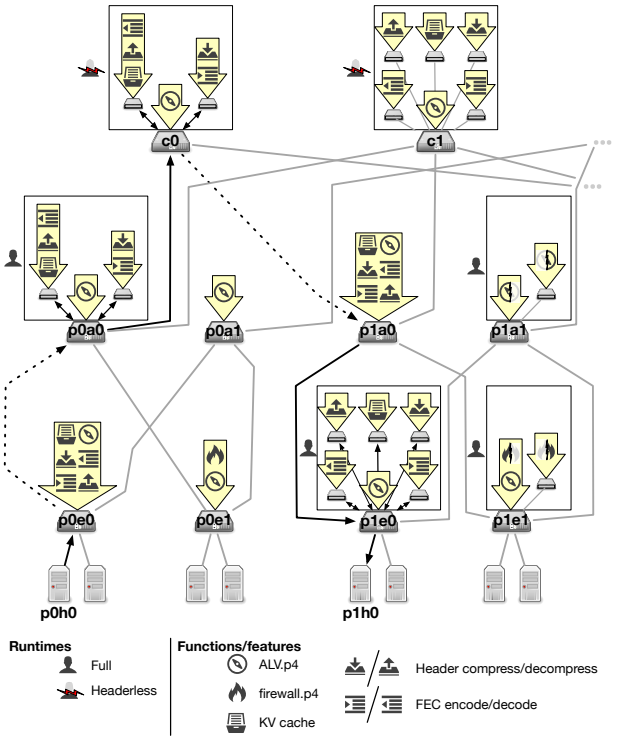


Figure 7: Part of our fat-tree network showing core routers (c*N*) and pods (p*N*[a,e,h]*M*) containing aggregation and edge switches, and hosts. Configured as explained in §7.1. Yellow arrows show which (sub)programs are installed on each device. Links between devices are shown in grey, except for links traversed by packets flowing between p0h0 and p1h0, which are shown in black. Dotted lines show faulty links.

switches and disaggregated them in various ways and to use different Flightplan runtimes, as described next. All disaggregated P4 programs, including those of third-party programs, were tested for behavioral correctness by checking that they produced similar results as the original programs.

7.1.1 Flightplan deployment example

Fig. 7 shows part of our network and the variety of P4 programs we ran simultaneously on different switches in the same network. Switches p0a1 (`ALV.p4`), p0e1 (`firewall.p4`), p0e0 and p1a0 (both `Crosspod.p4`) run non-disaggregated P4 programs, while all other switches run disaggregated programs. Of the latter, c0 and c1 use the headerless Flightplan runtime, and the remainder use the Full runtime.

Switches that run disaggregated programs are shown with an adjacent box in Fig. 7 showing the supporting devices on which parts of the original program were executed. For example, p1a1 carries out some of its table lookups locally, while others are offloaded to an associated device. Different switches can have different numbers of associated devices over which a dataplane program can be disaggregated—for

example, c0 has 2 while c1 has 5. The presence and resourcing of such devices is decided by the network operator. In this part of the evaluation we treat all such devices as being identical, but in the next section we distinguish between heterogeneous devices based on their resources and capabilities.

We also disaggregated `qos.p4` and `basic_tunnel.p4` and tested performance overhead in our Mininet-based setup. We found that the lower-bound overhead to client-perceived RTT was 8.2%. A more accurate measurement using a hardware-based experiment, but on a simpler topology, is given in §L.1.

7.1.2 Network Scalability and Operation

The network from Fig. 7 helps demonstrate two features. First, it shows how the use of Flightplan scales with network size. Since we constrain Flightplan's planning scope to only resources adjacent to a switch—as illustrated in Fig. 1—Flightplan's scalability is independent of network size in this network topology. Thus planning can scale to a network containing a large number of switches.

Second, it shows that different Flightplan runtimes and disaggregations can operate simultaneously in the same network, and that these can be configured and started independently of each other, to deliver the practical features described at the end of §2.

7.1.3 Overheads

We compare program-level overheads of dataplane disaggregations. These include overheads on: the network due to header inclusion, port count, data memory (total register bits, number of tables and their entries) and code memory (code from the runtime, and extra branching because of splitting). Device-level overheads—such as those on throughput and power—are evaluated in §7.2 based on testbed experiments.

Table 2 shows overheads for two sets of disaggregations of `Crosspod.p4`: one set using the Full runtime and the other using Headerless. In our prototype, the in-network programs do not respond to MTU path discovery, therefore to use the Full approach in general we needed to lower the MTU size to provide headroom for the Flightplan header. This uniformly reduces network capacity by 2.4% to create enough headroom.

Program-level overhead can be calculated before compilation, independent of toolchain. The general analysis of both runtimes' overheads is provided in §F. If a P4 program is split to run across several P4 devices, we can calculate the overhead introduced onto the network and the devices independently of the devices' toolchains.

7.2 Testbed Evaluation

We use a physical testbed deployment to measure device-level resource and performance impact of Flightplan-based disaggregation. This evaluation is based on the Headerless approach, which gives us a lower-bound on overhead among our runtime approaches. Using `Crosspod.p4` (§2), we: (i) evaluate the Flightplan planner, and (ii) measure the end-to-end performance and power consumption of a heterogeneous dataplane running Flightplan-generated splits of `Crosspod.p4`.

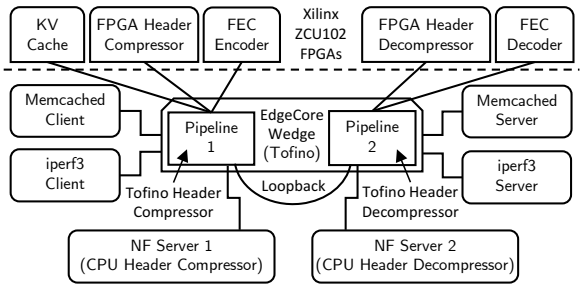


Figure 8: Evaluation testbed

7.2.1 Experimental setup.

Fig. 8 illustrates our testbed. It contains an EdgeCore Wedge 3.2-Tbps switch with a two-pipeline Barefoot Tofino P4 ASIC, five 4×10-Gbps Xilinx ZCU102 FPGA boards each with one Xilinx XCZU9EG FPGA, four traffic generation servers with 8-core Intel Xeon 2450 CPUs, and two network function servers with 10-core Intel Xeon Silver 4114 CPUs. In our configuration, the Tofino pipelines act as two independent switches, `S1` and `S2`, that service different ports and each have their own dedicated resources (SRAM, TCAM, etc.). All packets between `S1` and `S2` traverse a physical 10-GbE cable. We also evaluate a legacy scenario, by swapping the Wedge for an Arista 7050-QX32 with a Broadcom Trident II ASIC and OpenFlow support.

Crosspod.p4 plans. Our benchmarks focus on three plans output by Flightplan, designed for different objectives.

- **"Maximum Performance"** optimizes end-to-end performance by placing each function on the fastest available platform, i.e., header compression on the Tofino and all other functions on FPGAs.
- **"Resource Saver"** offloads compression from the Tofino to the FPGA to save compute and memory resources.
- **"Legacy Extender"** uses the Trident II in place of the Tofino to achieve functional equivalence with an OpenFlow switch.

Measurements. We benchmark four axes of performance: throughput, packet loss, latency, and power consumption. Throughput is measured at the application layer, using either `iperf3` or DPDK-pktgen. We measure packet loss using the Tofino's port counters. For latency, we use a simple telemetry function for the Tofino that timestamps (nanosecond precision) each packet ingressing or egressing out of monitored ports and clones a digest to a collection server. Finally,

	Feature	Runtime=**Full**									Runtime=**Headerless**								
Net.	Header (b)	288									0								
	Ports [*D*]	1-2 [3]			1-5 [6]						2 [3]			5 [6]					
	Ex. in Fig. 7	p0a0			p1e0						c0			c1					
	Seg ID	0	1	2	0	1	2	3	4	5	0	1	2	0	1	2	3	4	5
Dataplane	Tables	6			6						5	1	1	5	1				
	Entries (b)	179	89	89	445	89					154	22	22	230	22				
	Registers (b)	547	309	309	1261	309					4			4					
	Ctrl Struct	6	+2	+2	6	+2					2	+1	+1	2	+1				
	Externs	0	3	2	0	1					0	3	2	0	1				

Table 2: Overheads incurred by different disaggregations of `Crosspod.p4`, organized into **Network** and **Dataplane** overheads. The *Externs* row does not show overheads, but serves to show the distribution of extern function invocations across the splits. *D* is the number of segments. Each segment is given a separate identifier in the *Seg ID* row to distinguish them in the rows below. *Ports* is the number of switch ports that are required: the Full runtime's use of a Flightplan header allows the use of a single port (connecting to a single supporting device), while the Headerless runtime requires an exact number of ports. *Entries* gives the total size of all Flightplan table entries, in bits. *Ctrl Struct* is a measure of code complexity of the transformed program, counting the number of conditional statements introduced by the disaggregation in addition to the runtime's code.

we measure power consumption of each device at the outlet, polling at a 200ms interval.

End-to-end benchmarks. We use workloads that mix two types of flows: 1) large TCP flows (`iperf3`) representing traffic with high bandwidth demands and 2) UDP Memcached request streams representing latency-sensitive traffic. This workload models the bimodal distribution of packet sizes and traffic patterns in datacenters [4, 32]. We measure throughput using `iperf3` and calculate latency as the difference between the time at which a packet ingresses from its source host and the time at which it egresses to its destination host.[5]

7.2.2 Flightplan Planner

We used the Flightplan planner to analyze the solution space for `Crosspod.p4`. Our evaluation involves 239 rules, divided as follows: 140 are profile rules including Rules 1, 3, and 4; 68 are network rules (including those in Fig. 17), and the rest are program rules (including Rule 2).

The network is the one shown around `c1` in Fig. 7. We used this to explore program-segment mappings to the switch and its supporting devices in such an arrangement. We ran different variants of the experiment to contrast the implications of using different types of equipment, following the plans described in §7.2.1.

Fig. 9 summarizes our results, and we evaluate the best-performing plans on our physical testbed in the next section. To avoid clutter we exclude "Resource Saver" because of its hybrid nature between the other two categories, and instead we show more extreme forms of "Legacy Extender" in the "Server Offload" family. These rely on a single switch and a pool of CPU-based servers. The latency advantage of "Server Offload (Tofino)" relative to "Server Offload (Arista)" seems too optimistic, and we believe that might be because of a lack of accuracy in the model we use.

[5] As well as the end-to-end experiments on `Crosspod.p4`, we also separately evaluate its constituent in-network functions end-to-end in §A.

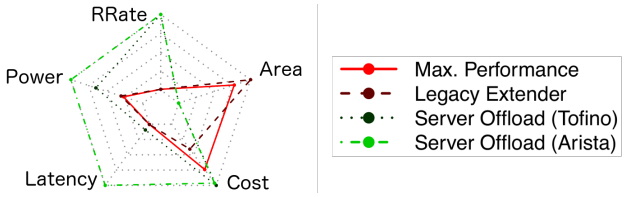


Figure 9: Best-rated plans found by Flightplan, described in §7.2.2. In each dimension, less is better. Each dimension is normalized by the maximum value from all plans. 'RRate' is the inverse of the bandwidth of the distributed program. 'Cost' refers to hardware costs and excludes running costs, which are captured by 'Power'. 'Area' refers to FPGA-resource usage.

7.2.3 Crosspod.p4 Benchmarks

The goal of `Crosspod.p4` is to improve the performance of applications bottlenecked by inter-rack links suffering from congestion and partial failure [39]. We evaluate how Flightplan's `Crosspod.p4` plans achieve this goal in a network with a 10 Gbps inter-rack link.

End-to-end performance. First, we evaluate the **"Maximum Performance"** `Crosspod.p4` plan. Fig. 10 shows application-level performance in a series of trials where the network functions are activated one by one. The figure plots the latency and success of GET and SET requests to the Memcached KV store and the throughput of TCP `iperf3` sessions, on two client/server pairs, as illustrated in Fig. 8.

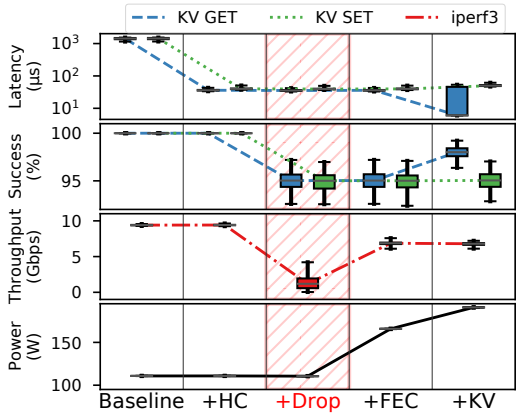


Figure 10: Throughput, latency and power utilization of a disaggregated `Crosspod.p4`. In the top and bottom panels, lower values are better. In other panels, higher values are better. Boxes show data quartiles and whiskers show 10th and 90th percentile, aggregated over 5 repetitions.

The leftmost panel in Fig. 10, labeled `Baseline`, shows application performance with no network functions enabled. Here, the problem is congestion, caused by the `iperf3` data transfers that saturate the inter-rack link. Because of the congestion, application-level Memcached latency is high.

To reduce the impact of congestion, we enable header compression of the `iperf3` traffic. This reduces the load on the bottleneck inter-rack link, allowing queues to drain and thus reducing latency. As the `+HC` column of Fig. 10 shows, header compression reduces Memcached request latency by 97% without impacting `iperf3` throughput. Since Flightplan maps the compression and decompression functions to the switch pipelines, no new network devices are required for this addition, and power consumption does not noticeably increase.

To study FEC, we first model partial link failure by enabling a dropping function in the switch that drops 5% of packets at random on the inter-rack link. The middle panel of Fig. 10, labeled `+Drop`, demonstrates application-level effect: a reduction in throughput to 1.19 Gbps and a loss of 5% of Memcached packets. The FEC encoder and decoder FPGAs are then introduced to protect TCP traffic on the link. It restores 57% of the lost TCP throughput, as the `+FEC` phase of Fig. 10 shows. Power consumption increases by about 55 W, reflecting the addition of the two FPGAs.

Finally, we introduce the Key-Value (Memcached) cache network function FPGA on the client side of the inter-rack link. The `+KV` column shows that the approximate 60% cache hit rate causes median latency of GET packets to reduce to 7 µs (while leaving the upper quartile above 40 µs) and restores successful responses to about 60% of previously dropped requests. SET latency and success are unaffected because the cache is inline, so these packets must traverse the shared link. Power consumption increases by another 25 W due to the addition of one more FPGA.

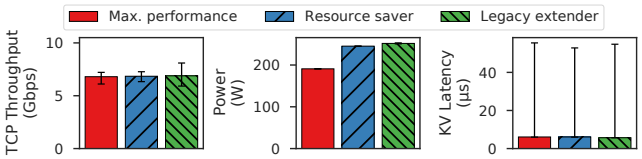


Figure 11: Application performance and network power consumption for three different `Crosspod.p4` plans. Bars mark medians, whiskers 5th and 95th percentile.

Alternate plans. Next, we evaluate the viability of other Flightplan-generated `Crosspod.p4` plans. As Fig. 11 shows, the **Resource Saver** and **Legacy Extender** provide almost identical application-level benefits. All three plans improve performance in the face of packet loss and congestion, increasing TCP throughput from 1.19 Gbps to 6.25 Gbps and decreasing median KV's GET request latency to under 10 µs.

Although all three plans are viable from the application's perspective, they offer different benefits for network operators. As Figure 11 shows, the power consumption of **maximum performance** is 50 W lower than the alternatives because it maps HC to the Tofino, thus requiring two fewer FPGAs.

On the other hand, by offloading HC to FPGAs, **Resource Saver** frees compute and memory resources in the Tofino. Most significantly, it reduces the number of pipeline stages from 10 to 4, the utilization of stateful ALUs from 25% to 2%, the number of tables from 46 to 11, and the utilization of SRAM by approximately 600KB per 1024 concurrent flows.

Finally, by targeting the Trident II, **Legacy Extender** lets OpenFlow switches achieve similar functionality and application-level performance as P4 switches. This provides a path to deploying `Crosspod.p4` in networks with limited programmability [32] or at a lower budget.

8 Related Work

Compared to Active Networking [2, 15, 33] and related recent work such as TPP [17], Flightplan leans towards performance and safety at the expense of flexibility: computation allocated to each dataplane is established at compile time, rather than being packet-carried.

Automated approaches to software splitting [6, 13, 22, 30] are usually done for vulnerability mitigation, whereas Flightplan is directed at improving performance and utilization. Closer to Flightplan, Floem [29] focuses on CPU-NIC co-processing of network applications, but Flightplan focuses on packet processing across distributed dataplanes which do not necessarily include CPUs.

As in network calculus [20] (NC) our reasoning technique for plans is concerned with flows at steady-state. Flightplan allows using arbitrary functions to describe the transformation γ of each rule on the abstract state, not only NC's operators. Future work can improve the modeling precision of our approach and generalize to better characterize workload-sensitive functions such as the in-network cache.

Acknowledgments. We thank Heena Nagda, Rakesh Nagda, our shepherd Nate Foster and the reviewers for their feedback. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contracts No. HR0011-19-C-0106, HR0011-17-C-0047, and HR0011-16-C-0056. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. *SIGCOMM Computer Communication Review*, 38(4):63–74, August 2008.

[2] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active Bridging. *SIGCOMM Computer Communication Review*, 27(4):101–111, October 1997.

[3] Barefoot Networks. Barefoot Tofino. https://www.barefootnetworks.com/technology/, 2016.

[4] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding Data Center Traffic Characteristics. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 65–72, New York, NY, USA, 2009. ACM.

[5] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[6] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[7] IOS Cisco. Quality of Service Solutions Configuration Guide. *Congestion Avoidance Overview. Cisco, Accessed*, 18, 2014.

[8] Carlos R Cunha, Azer Bestavros, and Mark E Crovella. Characteristics of www client-based traces. Technical report, Boston University Computer Science Department, 1995.

[9] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3):267 – 284, 1984.

[10] Facebook Inc. Introducing mcrouter: A memcached protocol router for scaling memcached deployments. https://bit.ly/2wDbLml, September 2014.

[11] D. C. Feldmeier, A. J. McAuley, J. M. Smith, D. S. Bakin, W. S. Marcus, and T. M. Raleigh. Protocol boosters. *IEEE Journal on Selected Areas in Communications*, 16(3):437–444, September 2006.

[12] Hans Giesen, Lei Shi, John Sonchack, Anirudh Chelluri, Nishanth Prabhu, Nik Sultana, Latha Kant, Anthony J McAuley, Alexander Poylisher, André DeHon, and Boon Thau Loo. In-network Computing to the Rescue of Faulty Links. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, NetCompute '18, pages 1–6, New York, NY, USA, 2018. ACM.

[13] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1016–1031, New York, NY, USA, 2015. ACM.

[14] Vishal Gupta, Ripal Nathuji, and Karsten Schwan. An Analysis of Power Reduction in Datacenters Using Heterogeneous Chip Multiprocessors. *SIGMETRICS Performance Evaluation Review*, 39(3):87–91, December 2011.

[15] Ilija Hadžić and Jonathan M. Smith. Balancing Performance and Flexibility with Hardware Support for Network Architectures. *ACM Transactions on Computer Systems*, 21(4):375–411, November 2003.

[16] Van Jacobson. Compressing TCP/IP Headers for Low-Speed Serial Links. RFC 1144, 1990.

[17] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. *SIGCOMM Computer Communication Review*, 44(4):3–14, August 2014.

[18] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[19] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, New York, NY, USA, 2010. ACM.

[20] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer Science & Business Media, 2001.

[21] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. ACM.

[22] Shen Liu, Gang Tan, and Trent Jaeger. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2359–2371, New York, NY, USA, 2017. ACM.

[23] Netcope Technologies. Netcope P4. https://www.netcope.com/en/products/netcopep4, June 2017.

[24] Netronome Inc. Agilio CX SmartNICs. https://www.netronome.com/products/agilio-cx/, 2016.

[25] P4 Language Consortium. P4 Behavioral Model. https://github.com/p4lang/behavioral-model, November 2018.

[26] P4 Language Consortium. P4 reference compiler. https://github.com/p4lang/p4c, November 2018.

[27] p4lang/tutorials. p4lang/tutorials. https://github.com/p4lang/tutorials.

[28] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 121–136, New York, NY, USA, 2015. ACM.

[29] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, Carlsbad, CA, 2018. USENIX Association.

[30] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[31] A.B. Roach. A Negative Acknowledgement Mechanism for Signaling Compression. RFC 4077, 2005.

[32] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. *SIGCOMM Computer Communication Review*, 45(4):123–137, August 2015.

[33] Beverly Schwartz, Alden W. Jackson, W. Timothy Strayer, Wenyi Zhou, R. Dennis Rockwell, and Craig Partridge. Smart Packets: Applying Active Networks to Network Management. *ACM Transactions on Computer Systems*, 18(1):67–88, February 2000.

[34] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Hong Liu, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. *Communications of the ACM*, 59(9):88–97, August 2016.

[35] Twitter Inc. Twemcache: Twitter Memcached. https://github.com/twitter/twemcache, June 2013.

[36] M. H. Van Emden and R. A. Kowalski. The Semantics of Predicate Logic As a Programming Language. *The Journal of the ACM*, 23(4):733–742, October 1976.

[37] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4FPGA: A Rapid Prototyping Framework for P4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 122–135, New York, NY, USA, 2017. ACM.

[38] Xilinx Inc. Xilinx SDNet. https://www.xilinx.com/products/design-tools/software-zone/sdnet.html, 2017.

[39] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and Mitigating Packet Corruption in Data Center Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 362–375, New York, NY, USA, 2017. ACM.

A Individual Function Evaluation

To validate their effectiveness, we measured the performance of individual in-network elementary functions—such as FEC, Memcached caching, and header compression—in our physical testbed. These functions are instances of *protocol boosters* [11].

A.1 Function 1: Forward-Error Correction

We developed a forward-error correction (FEC) link-layer network function [12] on FPGA to mitigate against corrupting links [39]. It introduces parity packets which are used downstream to recover corrupted packets. Our implementation uses a block code, supplementing every block of k (data) packets

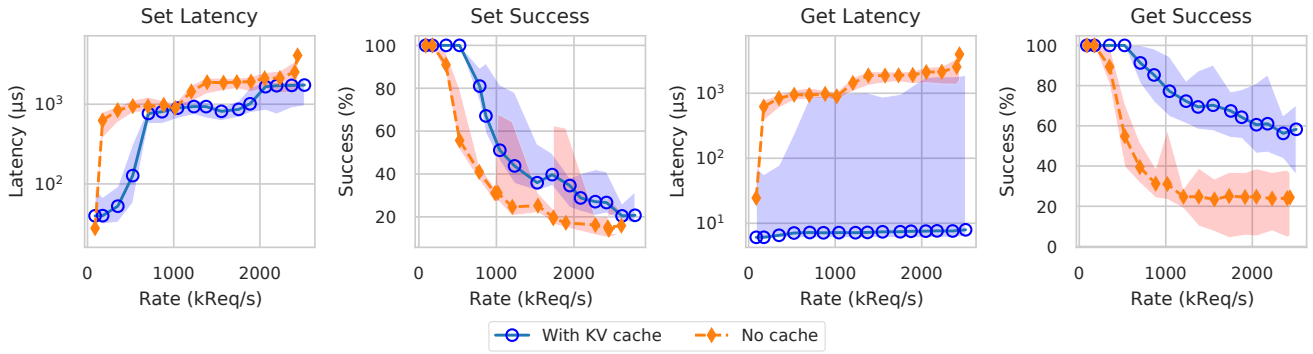


Figure 12: Performance metrics for Memcached with and without the KV-cache function. Markers show median values, while shaded regions show 5th and 95th percentiles. Success is measured as the percentage of responses received for every 500 requests.

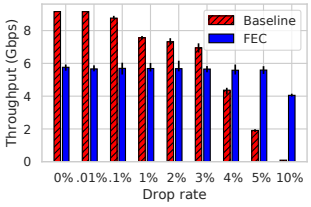


Figure 13: Effect of FEC on TCP throughput at varying rates of link loss

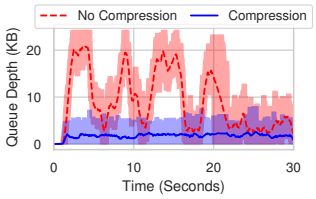


Figure 14: Effect of header compression on switch queue occupation

with h parity packets. The parity allows the in-network function to reconstruct up to h packets per block. The parity packets are computed by an implementation of the Reed-Solomon erasure code.

Forward-Error Correction reduces the need to retransmit packets in TCP flows and helps TCP sustain larger congestion windows across corrupting links. This is demonstrated by Figure 13 for an experiment performed with `iperf3` with 10 simultaneous TCP connections. Traffic was encoded with $k = 5$ and $h = 1$. Each bar represents the median throughput of 10 experiments, each of which lasted 1 min. Error bars show the minimum and maximum. For a packet drop rate of 5%, we observe that FEC increases the throughput by almost 3×. At 10% drop rate, throughput has all but vanished, but FEC manages to recover 40% of the 10-Gbps link capacity.

A.2 Function 2: Key-Value cache

We implemented an inline Memcached cache on FPGA to accelerate key-value queries. This improves the throughput, success rate, and latency of a Memcached deployment. The cache has the capacity to hold 1000 entries in its local hash table.

It was benchmarked against a standard Memcached server with default settings, including the use of four threads. Requests consisted of 8-byte keys and 512-byte values. 90% of requests sent were GET requests, while the remaining 10% were SET requests. Keys were chosen randomly from 10,000 candidates according to a Zipf distribution with an exponent of -1, consistent with measurements of document access frequency on the Web [8]. This distribution of request keys resulted in a cache hit-rate of around 50%.

Figure 12 shows the differential effects of the cache on SET and GET requests. As SET requests must still reach the Memcached server before they are acknowledged, the cache has minimal effect on SET packets, resulting only from the reduced load on the server. The inline cache keeps the median latency of GET requests below 10 µs regardless of the request rate. With the cache in place, approximately 50% less packets are lost than without the cache, consistent with its 50% hit rate.

A.3 Function 3: Header compression

We implemented an in-network function on the FPGA, Tofino and CPU for lossless compression of packet headers between neighboring switches. Our implementation is a simplified version of Van Jacobson compression [16] and many fixed-function implementations [7]. By compressing packets, bursts occupy less buffer space. Fig. 14 shows this effect for a highly-utilized (99%) network, as measured on the EdgeCore switch. Lines show the median, and the shaded region the range, over a rolling window of 100k samples.

B Single target micro-benchmarks

In addition to the end-to-end benchmarks of network functions, each individual device was also evaluated with fixed-rate packet replays using DPDK-pktgen.

Table 3 summarizes the maximum throughput and average latency that each function achieves on the CPU and FPGA targets.

Target	Function	Throughput (Mbit/s)	Latency (µs)
CPU	HC Compress	95.2 (0.038)	5800 (33.9)
CPU	HC Decompress	198.75 (1.5)	5900 (20)
FPGA	HC Compress	8350 (150)	5.24 (0.005)
FPGA	HC Decompress	8890 (17)	4.47 (0.002)
CPU	FEC Encode	35 (0.23)	300 (1400)
CPU	FEC Decode	64.47 (0.007)	90 (5.7)
FPGA	FEC Decode	7950 (27)	32.8 (1.4)
FPGA	FEC Encode	8130 (18)	4.75 (0.004)
FPGA	KV Inline cache	7730 (230)	15.9 (6.7)

Table 3: Throughput and latency values from per-function and per-target micro-benchmarks. Values shown are mean and standard deviation.

C Complexity Analysis of Dataplane Disaggregation

In this section we calculate the number of ways to split a program and allocate its splits to execute on nodes in the network subject to certain constraints. Starting with the number of ways to split a program, we abstract this as the number of contiguous subsequences $\{(1,\ldots,k_1),(k_1+1,\ldots,k_2),\ldots(\ldots,n)\}$ of a sequence $1,\ldots,n$ that represents the lines of the program. There are $(n-1)$ ways to bisect the sequence $1,\ldots,n$. Bisecting it again—to yield three subsequences—presents $(n-2)$ choices. In the general case, the number of k-ary dissections of an n-sequence (where $k<n$) is:

$$(n-1)\times(n-2)\times\ldots\times(n-k)= \left(\prod_{i=1}^{k}(n-i)\right)\in O\left(n^k\right) \quad (1)$$

Thus the number of ways to split a program grows exponentially in the number of splits sought.

Turning now to growth relative to the topology. Placement of subprograms occurs along a path between two hosts on the network. For simplicity assume that there is a single shortest path between the two, call it p. Let $A(p)$ be the set of vertices connected to p through a single edge, we call this the set of adjacent nodes. Subprograms can be placed along p or offloaded onto adjacent nodes drawn from $A(p)$. Since we are deriving an upper-bound, assume that each subprogram can be placed on any of the elements in $A(p)$ or placed on p. Let $|p|$ bet the length of path p, and $|A(p)|$ be the cardinality of set $A(p)$. The set of placement choices is $2^{|p|+|A(p)|}-1$, discounting the choice where no placement is made on any node. For a k-split program the number of placement choices is $\binom{2^{|p|+|A(p)|}-1}{k}=$

$$\prod_{i=1}^{k}\left(\frac{2^{|p|+|A(p)|}+1-i}{i}\right)\in O\left(\left(2^{|p|+|A(p)|}\right)^k\right)$$

Since in this theoretical model the choices of splitting and placement are independent then, modulo the simplifying assumptions, the space of disaggregation choices grows exponentially with the number of targets available on which to map subprograms, and doubly with the number of subprograms: $O\left(n^k\left(2^{|p|+|A(p)|}\right)^k\right)$. ■

Listing 2: Flightplan header definition

```
#define SEGMENT_DESC_SIZE 4
#define SEQ_WIDTH 32
#define STATE 8

header Flightplan_h {
  // Includes Ethernet header to simplify parsing, and handling by black-box
       external functions that aren't aware of the Flightplan header.
  bit<48> dst;
  bit<48> src;
  bit<16> type;

  bit<SEGMENT_DESC_SIZE> from_segment;
  bit<SEGMENT_DESC_SIZE> to_segment;
  bit<STATE> state;
  bit<BYTE> byte1;
  bit<BYTE> byte2;
  bit<BYTE> byte3;
  bit<BYTE> byte4;
  bit<QUAD> quad1;
  bit<QUAD> quad2;
  bit<QUAD> quad3;
  bit<SEQ_WIDTH> seqno;
}
```

D Flightplan header

Listing 2 shows the P4 definition of the header which Flightplan uses when sending packets between connected dataplanes.

E Runtime Implementations

This section elaborates on the description provided in §5 of our implemented Flightplan runtimes and their resource overheads.

E.1 Full Runtime

This section expands the description given in §5.3.

To detect network distortion such as drops, reordering, and duplication, our scheme includes a sequence number in the Flightplan header. A corresponding amount of state is kept at the sending and receiving dataplanes to determine whether the sequence of packets has been interrupted. Negative acknowledgment [31] is used for a downstream dataplane to signal loss of synchronization with the upstream dataplane. Positive acknowledgments are used by the upstream dataplane to poll liveness of the downstream dataplane.

This mechanism sets up a feedback loop between connected dataplanes. Our scheme is illustrated in Figure 15. Using this scheme we push fault detection into the dataplane, to react to faults in the network by triggering a relink or escalating a notification.

Among our runtimes, Full provides the most flexibility when splitting a program: control-flow can be stopped anywhere in the program and resumed later on a different dataplane using context that was serialized into the Flightplan header. Flightplan has no visibility into dependencies on state that is controlled from outside the dataplane, such as extern

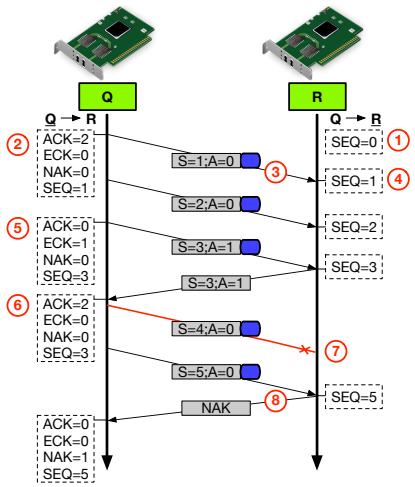


Figure 15: ① Synchronization state is initialized at downstream dataplanes. ② Synchronization state is initialized at upstream dataplanes. ③ Synchronization information is included in the Flightplan header (shown in grey) that is added to packets (shown as the blue square). ④ Sequence number is incremented. ⑤ Periodically upstream dataplane will poll downstream dataplane for activity, by raising the ACK bit in the Flightplan header, and the ECK (Expecting ACK) locally. ⑥ ECK is reset when an ACK packet is received. ⑦ Packets may be lost in either direction, leading to loss of synchronization. ⑧ Loss of synchronization is eventually detected and action is taken. Negative acknowledgment (NAK) seeks to update the upstream dataplane.

function state and tables; those need to be synchronized externally.

The runtime keeps track of the following sets of information: N (the *next* dataplanes that a dataplane might transfer to), P (*previous* such dataplanes), S_N (which fail-over alternative to use for each next-dataplane), and N_F and P_F (metadata for failure detection and handling, such as sequence numbers and whether an acknowledgement has been demanded).

On each dataplane on which part of the disaggregated program is run, `fpctl` configures dataplanes to fail-over to use other downstream dataplanes in the event of reaching a threshold of received NAKs or missed ACKs. Both thresholds are absolute values (not ratios, for instance) that are configured by `fpctl`, which can also periodically poll the state of each runtime to determine if it has seen an increase in failures. Various actions can be taken depending on such an observation, for example: i) the threshold could be raised—again, using `fpctl`—to buy time to understand the source of the loss, ii) rate limits could be applied to applications using the link, iii) the state and configuration of the downstream dataplane can be inspected or reset, iv) regardless of whether the cause has been understood or not, the ACK and NAK counts could be reset periodically if they are no longer seen to increment regularly. These actions are not directly implemented in

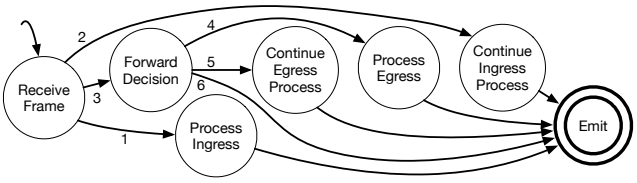


Figure 16: Core logic of the Headerless runtime.

`fpctl` but they could be automated by the network operator by using the functionality provided by `fpctl`.

If no further fail-overs are possible, then the dataplane shuts down; in turn, upstream dataplanes will eventually fail-over to use a different dataplane, if possible, when their threshold is met.

E.2 Headerless Runtime

The Headerless approach works by building a circuit between dataplanes. Compared to the Full approach, in which a program can be split almost arbitrarily, Headerless provides less flexibility.

A program is assumed to have three parts: the ingress subprogram, the forwarding subprogram, and the egress subprogram. The ingress or egress subprograms, or both, may be empty. If non-empty, they are segmented for offloading into supporting devices, which we call *helper* dataplanes.

Thus the HL runtime differentiates starkly between two types of dataplanes: the switch, on which the forwarding subprogram is kept, gets the HL_S version of this runtime, while the helpers get the much simpler HL_H part. The HL_H part of the runtime receives offloads from HL_S, performs a computation, and returns control back to HL_H. Disaggregations using HL have a single instance of HL_S and potentially several of HL_H. The ingress and egress subprograms, if non-empty, are executed on a dataplane running HL_H.

Fig. 16 shows the core logic of this circuit, which takes place on HL_S. It reflects the three parts in which the program is organized: first, one or more ingress subprogram segments (I_N) are executed, after which a forwarding step is made to determine whether to execute the egress segments (E_N). Upon arrival, if the ingress port is in I then transition 1 is taken, else if the port is in I_N then transition 2 is taken. Otherwise a forwarding decision is made, and if the egress port is in E then transition 4 is taken, or if the port is in E_N then transition 5 is taken; otherwise transition 6 is taken if the program's egress segment does not apply to the egress port.

A consequence of using the Headerless runtime is that disaggregation must be coarser—only linear segments are possible, not at DAG, and downstream live-variables must be included in the same segment that updates the variables. As a result, the complexity of segmenting is strictly lower than the general case, as shown in §H. The suitability of using Headerless runtime with a given segmentation is detected statically using our P4 compiler extension.

F Runtime Overhead

Table 4 accounts for the overheads of each approach. The overheads are described as functions over runtime-specific parameters described in §E.

The constants in each function were derived by counting the state used in each runtime, such as the width of registers, and the width of entries from a table's P4 specification. Further, we counted the number of table entries resulting from typical configuration of each runtime.

For example, the function $N \cdot 13$ for HL(IPv4)'s total size of entries (in bits) is derived from the number of entries N—the number of next dataplanes that a segment might forward to—which is known at compile time, multiplied by 13 which consists of 9 bits for the port number and 4 bits for the segment number. The state used by register arrays is the product of the array size and the register size. For example, the term $(N_F + P_F) \cdot 228$ contributing to Full's register overhead consists of register arrays whose register sizes total 228 bits, one of which is of size N_F and the other P_F. The parameters are described in the previous section and in the table's caption.

These overheads are not program-specific, but rather they are additional to the program's overheads. Also, these overheads are per-dataplane, not per-program. So if a dataplane program is disaggregated into more subprograms then it will incur more cumulative overhead, but the overhead's factor will depend on the runtime involved.

So for example if an L-line P4 program used R register bits and T tables, whose entries occupied $|T|$ bits, were disaggregated into N subprograms, then if it were to use the Headerless runtime, each subprogram S would be of size $L_S + 70$, where $\frac{L}{N} \approx L_S \leq L$ is the subprogram's line count. HL_S would have $R+4$ register bits, $T+5$ tables. Assuming that $I = E = 1$, and that half of the N subprograms relates to the ingress segment (and therefore the other half relates to the egress segment), we have that $I_N = E_N = \frac{N}{2}$, and therefore the memory needed for table entries is

$$\begin{aligned} & |T| + \left(22 + \tfrac{N \cdot 22}{2} + 22 + \tfrac{N \cdot 22}{2} + 10\right) \\ = \quad & |T| + N \cdot 22 + 54 \end{aligned}$$

bits. The overhead of HL_H can be calculated similarly.

G Complexity Analysis of Headerless Disaggregation

Below we define the complexity function C of headerless disaggregation, in terms of the number of possible choices that can be made during this process. The asymptotic complexity of disaggregating a program to use the Headerless runtime is $\max(C(S_I), C(S_E))$, where S_I is the sequence of lines in the ingress segment, and S_E the sequence of lines in the egress segment.

Let S be a sequence of lines of code, $n = |S|$ be the number of lines in S, and $1 \leq B \leq n$ the arithmetical average, in lines of code, of blocks in S. Top-level lines of code that do not form part of a block are counted as blocks of length 1. The number of blocks in S is $1 \leq \frac{n}{B} \leq n$.

At the limit, each block will be mapped to a different segment. But there might be constraints that force us to merge segments or objectives that are benefitted by such merges. Let $M \in \mathbb{N}, 0 \leq M \leq \left(\frac{n}{B} - 1\right)$ be the *merge opportunity*: i.e., the number of merges of (adjacent) segments we can carry out to the maximally segmented program. Increasing M has the effect of making segmentation coarser by merging more blocks together. Blocks are merged to eliminate data-flow dependencies between blocks since the Headerless runtime does not allow context serialisation and transfer between dataplanes. Blocks are also merged to make better use of physical resources; in our setup this merge decision is made by the planner.

The value of C consists of the sum of merge choices for each value of M, which by the binomial theorem simplifies to an exponential function:

$$C(S) = \sum_{M=0}^{\frac{n}{B}-1} \binom{\frac{n}{B} - 1}{M} = 2^{\frac{n}{B}-1} \in O\left(2^{\frac{n}{B}}\right) \tag{2}$$

■

H Comparative complexity

In this section we show that for programs having more than 2 lines, our Headerless disaggregation (§G) presents fewer choices than general disaggregation (§C).

We start by clarifying the number of choices presented by general disaggregation. From Equation 1 in §C, the complexity of general disaggregation is $O\left(n^k\right)$. Recall that $k < n$. Expanding for all possible values of k, which represents the number of splits we may choose, we obtain:

$$O\left(n^1 + \cdots + n^{n-1}\right) \subseteq O\left(n^{n-1}\right)$$

From Equation 2, the complexity of Headerless disaggregation is $O\left(2^{\frac{n}{B}}\right)$.

Assuming $n > 2$, our goal is to show $O\left(2^{\frac{n}{B}}\right) \subset O\left(n^{n-1}\right)$.

Recall that $1 \leq B \leq n$, and note therefore that $O\left(2^{\frac{n}{B}}\right) \subseteq O(2^n)$.

Without changing our assumption regarding n, we use the transitivity of $\subseteq$ to reformulate our goal as $O(2^n) \subset O\left(n^{n-1}\right)$. Take 2^n and n^{n-1} as the bounding functions of the two classes respectively. We show that $2^n < n^{n-1}$ first through application

[6] In addition to the resources quantified in Table 4, the Full approach also uses mirroring sessions to provide feedback. One session is used for each upstream dataplane to which it needs to provide feedback. These are configured automatically by `fpctl`.

[7] The header can be enlarged to afford more scratch space if further headroom is made available from the interfaces' MTU.

[8] We repurpose the Fragment Offset field instead of adding a header.

Runtime	#LOC	#Tables	Entries(#bits)	Registers(#bits)	Header(#bits)
Full[6]	400	6	$N \cdot S_N \cdot 17 + 2N \cdot 36 + 2N_F \cdot 36 + 2P_F \cdot 40$	$71 + (N_F + P_F) \cdot 228 + N \cdot 10$	288[7]
HL(IPv4)	75	1	$N \cdot 13$	11	0[8]
HL $\begin{cases} \text{HL}_S \\ \text{HL}_H \end{cases}$	70	5	$I \cdot 22 + I_N \cdot 22 + E \cdot 22 + E_N \cdot 22 + 10$	4	0
	70	1	22	4	0

Table 4: Dataplane and network overheads for each runtime. In HL we differentiate between HL_S and HL_H: the switch dataplane and the helper dataplanes. These overheads were calculated by counting the resources used by each runtime's P4 implementation, and they are inherited by every dataplane involved in running a disaggregated program that uses that particular runtime. N is the number of *next* dataplanes that a dataplane might transfer to, P is the number of *previous* such dataplanes. $S_N \geq 1$ is the number of fail-over states: if $S_N = 1$ then there is no fail-over. $N_F \leq N$ and $P_F \leq P$ is the number of next and previous dataplanes for which a *feedback loop* is configured. I and E are the number of ingress and egress ports for which a program is running, and $I_N \geq 0$ and $E_N \geq 0$ are the number of intermediate hand-overs during the ingress and egress stages of the program.

of identities:

$$\begin{aligned} & 2^n < n^{n-1} \\ \Leftrightarrow\ & \log_2(2^n) < \log_2(n^{n-1}) \\ \Leftrightarrow\ & n\log_2 2 < (n-1)\log_2 n \\ \Leftrightarrow\ & n < (n-1)\log_2 n \\ \Leftrightarrow\ & 0 < (n-1)\log_2 n - n \end{aligned}$$

Then by induction on n, using $n = 3$ for the base case, verifying that $0 < 2 \cdot 1.58 - 3$, and then assuming the induction hypothesis $0 < (n-1)\log_2 n - n$ to show $0 < n\log_2(n+1) - (n+1)$. We start with application of identities:

$$\begin{aligned} & 0 < n\log_2(n+1) - (n+1) \\ \Leftrightarrow\ & 0 < n(\log_2 n + \log_2\left(\tfrac{n+1}{n}\right)) - (n+1) \\ \Leftrightarrow\ & 0 < n\log_2 n + n\log_2\left(\tfrac{n+1}{n}\right) - n - 1 \\ \Leftrightarrow\ & 0 < \log_2 n + (n-1)\log_2 n + n\log_2\left(\tfrac{n+1}{n}\right) - n - 1 \\ \Leftrightarrow\ & 1 < ((n-1)\log_2 n - n) + \log_2 n + n\log_2\left(\tfrac{n+1}{n}\right) \end{aligned}$$

Using the induction hypothesis we conclude that $((n-1)\log_2 n - n) > 0$, and using the assumption that $n > 2$ it follows that $\log_2 n > 1$ from the definition of $\log_2$, and $\log_2\left(\frac{n+1}{n}\right) > 0$ from the definition of $\log_2$ and since $\frac{n+1}{n} > 1$. ■

I Modelling Language

We use a simple formal language to describe relations between segments, computational resources and network resources. The relations rely on symbols used to represent those entities, such as "Compress", "PacketSize", and "FPGA". We require the user to declare these symbols by specifying a *signature*. A Flightplan signature consists of two finite sets: propositions Prop and variables V.

Signatures can be reused, at least partly, by different programs on the same network, or by the same program being deployed to different networks. Our P4 compiler extension generates an initial signature together with the abstract programs and the initial resource rules. The user can then extend and maintain this as needed.

$$\frac{\text{FPGA} \quad \langle \text{Rate} < 9.5 \times 10^9 \rangle \quad \text{PacketSize} > 100}{\text{header\_compress}} \left[\begin{array}{l} \text{Lat.} \mapsto \text{Lat.} + 6.44 \times 10^{-6} \\ \langle \text{Rate} \mapsto \text{Rate} \times \frac{9.15}{9.3} \rangle \\ \langle \text{LUTs} \rangle \mapsto \text{LUTs} + 24.4\% \\ \langle \text{BRAMs} \rangle \mapsto \text{BRAMs} + 54.4\% \\ \langle \text{FF} \rangle \mapsto \text{FF} + 15.8\% \\ \text{once Power} \mapsto \text{Power} + 30\,\text{W} \\ \text{once Cost} \mapsto \text{Cost} + 2 \end{array}\right]$$

Rule 4: Running Header Compression on an FPGA. Angled brackets indicate that the variables they reference depend on the device—e.g., instead of updating the flip-flop (FF) count for all FPGAs, we increment those of the FPGA to which HC is mapped using this rule.

$$\frac{\text{CPU}}{\text{egress\_compression}}\ [\text{Id}]$$

Rule 5: Flightplan's static analysis identifies egress_compression as a resource, and specifically as a table. We add a rule that allows this resource to be used on a CPU-based target.

Resource-related syntax is mapped to distinct propositions from Prop. We encountered the proposition `header_compress` in Rule 1. That rule had another proposition, "CPU", that is external to the program but is used to qualify the semantic to a specific hardware.

The variables in V are used to construct a second syntactic category in our specification language called Bound. For each $v \in V$ and $r \in \mathbb{R}$, the following are valid bounds: $v \,\text{op}\, r$ for $\text{op} \in \{=, <, >, \leq, \geq\}$. Rules 1 and 4 show examples of bounds usage from our formalization.

Propositions and bounds are also used to express constraints of network hardware. The planner (§6) uses these rules to explore implications of using those devices in plans. Fig. 17 shows a subset of the rules used for the evaluation described in §7.2.2, which model the network described in §7.2.1.

$$\frac{\text{CPU}}{\text{drop}}\ [\text{Id}]$$

Rule 6: Flightplan's static analysis identifies drop as an external function. We add a rule that allows this to be called on a CPU-based target.

$$\begin{aligned}
&\text{edgecore-2}: \\
&\quad \pi_{\text{Requires}} = \{\text{Rate} \leq 10^{11}\}, \\
&\quad \pi_{\text{Provides}} = \{\text{PSwitch}\} \\
&\text{arista-1}: \\
&\quad \pi_{\text{Requires}} = \{\text{Rate} \leq 4 \times 10^{10}\}, \\
&\quad \pi_{\text{Provides}} = \{\text{Switch}\} \\
&\text{ZCU102-5}: \\
&\quad \pi_{\text{Requires}} = \{\text{Rate} \leq 10^{10}\}, \\
&\quad \pi_{\text{Provides}} = \{\text{FPGA}\} \\
&\text{Xeon2450-1}: \\
&\quad \pi_{\text{Requires}} = \{\text{Rate} \leq 10^{10}\}, \\
&\quad \pi_{\text{Provides}} = \{\text{CPU}\}
\end{aligned}$$

Figure 17: A subset of the network-description rules used for the evaluation described in §7.2.2. Unlike other rules we have seen, these rules describe the constraints on using each device and the features enabled by each device. The planner uses this information when exploring the space of solutions.

J Generating rules from empirical profiles

Table 3 provided a summary of our profiling experiments. This section describes how the profiling experiments were done, and how their results were used to generate the *profile-related rules* such as Rules 1, 3 and 4. Other types of rules are derived from the program or from the network's description and are generated differently as described in §4.

J.1 Methodology

For each external function involved in a program, we measure its peformance characteristics on different classes of devices and use this information to estimate the performance of program segments in which those functions occur.

The performance profile is made by installing the function on a member of each class on which it can run and running a workload to sample the function's performance. This is used to create an entry in a table consisting of the following columns:

$$\begin{aligned}
&(\text{Function}, \text{PacketSize}, \text{Target}, \text{Time}_{\text{Arrive}}, \text{Time}_{\text{Leave}}, \\
&\quad \text{Rate}_{\text{Arrive}}, \text{Rate}_{\text{Leave}})
\end{aligned}$$

The first three columns consist of the function's name, the packet size used in the workload, the name of the hardware target class. The next two columns consist of the timestamps of when a frame is received by the function, and the processed frame is sent by the device.[9] Similarly, the last two columns consist of the arrival rate to the function and the sending rate. This captures the effect of the function on the link's capacity: for example FEC uses more of the downstream capacity compared to upstream, while header compression uses less.

Further, we also make two measurements that are less function- and workload-dependent: Power consumed by the target when executing the function on that workload, and the Cost of the hardware target. Instead of literal equipment costs we used relative quantities: $\{\text{FPGA} = 1, \text{Switch} = 2, \text{CPU} = 5, \text{PSwitch} = 10\}$.

Limitations on precision. This approach will not accurately account for differences between devices in the same class—e.g., different FPGA devices—or different configurations of the same device—e.g., kernel bypass on CPU targets—but it will give us good-enough characterization for our purposes, and a starting point for more accurate characterizations. Like all profiling, it is also sensitive to the workload used. Despite this, our approach is amenable to refinement in the following way: devices classes can be refined into subclasses, different kinds of workloads can be distinguished by adding a variable to each rule, and additional variables can be sampled by measurements and added to the generated rules. For example, the $\langle \cdots \rangle$ in Rule 4 involves an extension we made to more accurately characterize the usage of FPGA resources. Since the utilization of FPGA resources is a simple additive approximation we do not constrain the planner by their values—for instance to ensure that an FPGA's LUTs are not exceeded—but instead we do a post-hoc check.

Limitations on scale. To scale with the size of the network, we avoid creating a profile for every target in the network. Further, to scale with program and disaggregation diversity, we avoid creating a profile for every segment. Instead we focus on external functions, and measure their performance characteristics on different classes of devices as described above. In future work some of this characterization work can be automated further, to improve both its scalability and its precision.

J.2 Compiling the table

For each Function and Target we run a series of experiments in which we gradually increase PacketSize and $\text{Rate}_{\text{Arrive}}$. As long as we do not notice any drops—which would indicate that the function is not coping with the arrival rate for that packet size—we create a table entry for the information described above.

We then process the raw table to create a second table consisting of the following columns, some of which overlap with the first table we described:

$$\begin{aligned}
&(\text{Function}, \text{PacketSize}, \text{Target}, \Delta\text{Latency}, \text{Rate}_{\text{Arrive}}, \Delta\text{Rate}, \\
&\quad \text{Power}, \text{Cost})
\end{aligned}$$

[9] Both timestamps were generated by the EdgeCore Wedge switch and were mirrored to a collection server on the side.

$$\frac{\begin{array}{l}\text{Function} \quad \text{Rate} < \left(\text{Rate}_{\text{Arrive}} + T_{\text{R}}\right) \\ \text{PacketSize} > \left(\text{PacketSize} - T_{\text{PS}}\right)\end{array}}{\text{Target}} \left[\begin{array}{l}\text{Latency} \mapsto \text{Latency} + \Delta\text{Latency} \\ \text{Rate} \mapsto \text{Rate} \times \Delta\text{Rate} \\ \text{once Power} \mapsto \text{Power} + \text{Power} \\ \text{once Cost} \mapsto \text{Cost} + \text{Cost}\end{array}\right]$$

Rule 7: Template for performance-related rules. T_{R} and T_{PS} are tolerance offsets for rate and packet-size respectively. We set these to small values to retain fidelity while using the inequality operators.

where

$$\begin{aligned}\Delta\text{Latency} &= \text{Time}_{\text{Leave}} - \text{Time}_{\text{Arrive}} \\ \Delta\text{Rate} &= \frac{\text{Rate}_{\text{Leave}}}{\text{Rate}_{\text{Arrive}}}\end{aligned}$$

All the records from the first table are automatically processed to create the second table. This is an example record from which we will show how to generate Rule 1:

$$\left(\text{header\_compress}, 1000\,\text{B}, \text{CPU}, 7.4 \times 10^{-3}\text{s}, 2 \times 10^{8}\text{Gbps}, \frac{189.9}{194.75}, 150\,\text{W}, 5\right)$$

J.3 Converting table into rules

Finally, the table of measurements described above is converted into rules. This is done automatically as follows: for each row in the second table we instantiate the template shown in Rule 7 by replacing the black-background fields in the rule with the corresponding fields in the row. The meaning of these rules is explained in the caption of Rule 1 in §3, and in §I.

That is how we generated the performance-related rules used by our tools, including the following rules shown in this paper: Rules 1, 3 and 4.

K Program Annotation & Splitting

After finding a mapping from segments to dataplanes as part of a plan, contiguous segments that are mapped to the same dataplane are grouped into a single segment. This is done by deleting intermediate `flyto()` statements. At this point the program is said to be annotated for splitting.

Next we need to generate a physical, well-formed and self-contained P4 file for each segment. The program splitting phase performs three tasks: 1) extract the P4 code from each segment, forming subprograms; 2) analyze the subprograms to gather runtime-related context, such as variables whose values must be included in the hand-over between dataplanes; 3) inject runtime-dependent code for handing-over between subprograms.

These tasks involve simple but laborious analysis and modification of data structures in the compiler, which we outline in this section.

Subprogram generation Our implementation reuses the P4 compiler's data structures as much as possible, adding a thin layer to facilitate the analysis and transformation needed for splitting. We follow these steps: **1)** Analyze the program's abstract-syntax tree (AST) to identify all occurrences of calls to `flyto`. **2)** Generate a *virtual AST* (vAST) for each `flyto`. A vAST is an overlay of the AST for each split. Thus we obtain a single data structure using which we can simultaneously reason about the original program and all derived programs. **3)** Generate the *segment-level topology*: a graph where nodes consist of segments and where an edge denotes a code-path leading to the hand-over between adjacent segments.

Context gathering The synthesis of code between segments to pass data across dataplanes involves computing the transitive closure of required state, to ensure that downstream dataplanes shall have all the information they need to compute on. Writing such interfacing code by hand is tedious and error-prone. We continue the steps taken in the compiler extension: **4)** Traverse each vAST to compute its set of free variables (variables that are used, but not defined, in that vAST); **5)** Traverse the segment-level topology to compute the *extended scope of variables*: that is, variables defined in one vAST will need to be propagated to downstream vASTs in which they are used.

Code injection, Flightplan header This task uses the information gathered during the previous stage to ensure that the hand-over will happen correctly between all connected subprograms: **6)** If supported by the runtime, then generate stubs to propagate two kinds of information across logical dataplanes: a) the values of extended-scope variables and b) additional Flightplan metadata for instrumented programs or fault detection (§5). This is added to a special Flightplan header that encapsulates the original packet, piggy-backing logistical metadata.[10] **7)** Embed the vAST in the original program, replacing the main control block. The downstream compiler will purge unnecessary declarations and definitions. For each vAST, we update the deparsing stage to use the stub for the downstream logical dataplane. We also update the parser block to opportunistically parse the Flightplan header, and, if successful, then the control block branches to process the continuation of Flightplan code.

L Disaggregation microbenchmarks

L.1 Latency Microbenchmarks

In this section, we measure the amount of latency added by splitting a dataplane and benchmark the responsiveness of Flightplan's failover mechanisms.

End-to-end latency. We benchmark application layer end-to-end latency by measuring round-trip time (RTT) between two servers connected to switch `S1` in the topology described in §7.2.1. A concurrent ping and 10 `iperf3` streams went from the `iperf3` client to the Memcached client, with the forwarding table initially stored on `S1` and offloaded to `S2`. Fig. 18 shows the change in ping RTT distribution. Latency

[10] The Flightplan header is described in Appendix D.

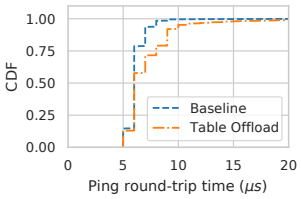


Figure 18: Table-offload overhead measured by pings across end-hosts.

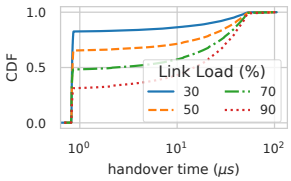


Figure 19: Effect of network load on handover latency.

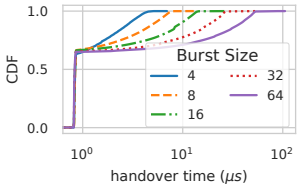


Figure 20: Effect of burstiness on handover latency.

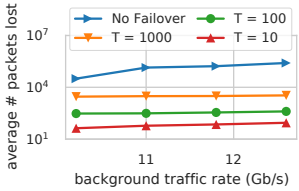


Figure 21: Thresholding (T) of packet loss.

was higher because packets had to traverse two additional queues in each direction. These queues were congested due to `iperf3`, which adds latency as discussed in Sec. 7.2.3. In this experiment we measured an 18.7% increase in average RTT and 22.3% increase in maximum RTT in our setup, but this increase will be smaller when the communicating servers are separated by more hops, for example if they are not both connected to `S1`.

Handover latency. To understand overhead at a finer scale, we measure *handover time*, the time between the invocation of the function containing the offloaded table and the beginning of its execution. Fig. 19 shows the distribution of handover time as link utilization varies. At a utilization level of 30%, reflecting high load in a data center scenario [4], handover time was under 1 µs for over 80% of the traffic. At higher rates, handover time increases due to added congestion in the queues between `S1` and `S2`. As Fig. 20 shows, larger packet bursts also have an effect, but only stretch the tail of the handover time distribution. In a 50% load scenario, the handover time remained under 1µs for over 60% of packets.

L.2 Failover Microbenchmarks

We evaluate two kinds of fail-over mechanisms. The first involves controller-based failure-detection and effecting of fail-over. The second involves in-dataplane failure-handling.

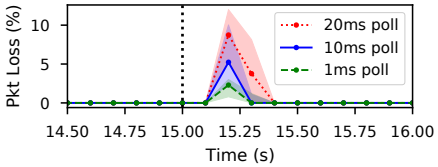


Figure 22: Packet loss during failover, as measured by `iperf3`. At time 15 s, the FEC encoder fails and is automatically replaced by another FPGA. Polling for detection of the failed link at tighter intervals results in less loss during the failover procedure.

A deployment can involve both simultaneously, with the in-dataplane mechanism reacting more quickly in most cases, and the controller-based approach handling cases where the in-dataplane mechanism is incapacitated because of device failure.

Controller-directed failover. In the case of total failure of a link or device, the loss of connection can be directly detected by the controller, allowing it to automatically initiate the failover procedure without further intervention.

Fig. 22 shows the packet loss rate of a UDP `iperf3` session running at 1 Gbit/s through a dataplane employing FEC during the failure of the FEC encoder FPGA. At time 15 s, the link to the FPGA is disabled. The controller, which polls for the presence of the link at regular intervals, redirects traffic to a failover FPGA once the down link is detected.

With a polling interval of 1 ms, at most 3% of packets are lost in the 100 ms interval immediately following the disabling of the link.

Dataplane NAK Failover. We benchmark the NAK mechanism described in §5.3 with a simple program on `S1` that offloads a no-op function `F`. There are two instances of `F` (`F.1` and `F.2`) that both run on switch `S2`, but service different links connected to `S1`. We measure the number of packets lost in a scenario where failover from `F.1` to `F.2` occurs due to congestion on the link to `F.1`.

Fig. 21 shows how the number of packets lost varies with the rate of congestion-inducing background traffic. The Flightplan failover mechanism bounds packet loss to a low (< 10) integer factor of the NAK threshold (T). After the Tth NAK is received, only in-flight packets already queued to `F.1` are lost because every subsequent packet is routed to the failover instance. Without the NAK mechanism, the number of packets lost is unbounded.