Work in Progress paper:

Experiment Planning for Heterogeneous Programmable Networks

Nik Sultana
Dept. of Computer Science
Illinois Tech
Chicago, Illinois

Abstract—**Private and publicly-funded cloud infrastructure and testbeds increasingly feature programmable network hardware. Programmable network cards and switches support the execution of increasingly-complex *in-network programs* that can operate independently of end-hosts to improve the network's performance, resilience and utilisation. Reasoning about in-network programs, their placement, and workloads is needed to plan jobs on programmable networks. On programmable testbed networks, this reasoning feeds into resource allocation, fairness and reproducible research. But this reasoning is made challenging by the performance and resource diversity of hardware and by the failure modes that can arise in a distributed system.**

Flightplanner is currently the most comprehensive reasoning system for distributed and heterogeneous in-network programs but it uses a custom formalism and tool implementation, making it difficult to understand, extend, and scale.

This paper describes Lightplanner, a generalisation of Flightplanner's reasoning system that has been implemented on Prolog. It provides an executable formalisation in a well-understood logic. By relying on Prolog's proof search, Lightplanner is $10\times$ smaller than Flightplanner's implementation in C++, making it better suited for others to understand, extend, and scale. A benchmark of publicly-available in-network programs is used to evaluate Lightplanner against Flightplanner. Though the time overhead is slightly larger, Lightplanner can find better allocations than the original, more complex C++ implementation.

Lightplanner is being incubated to plan experiments in a local programmable network testbed at Illinois Tech, and as a future step it will be extended to work across federated networks such as FABRIC.

Index Terms—**Programmable Networking, Resource Allocation, Program Analysis**

I. Introduction

The development of the *in-network computing* [1] paradigm is being propelled by the increased programmability of commodity network cards [2]–[5] and switches [6]–[8], and the increased traction of domain-specific languages like P4 [9]. In this paradigm, computations may occur at high throughput and low latency inside the network—on network switches and network cards (NICs), independently of end-hosts. This paradigm supports operational services like caching [10] and monitoring [11], and is the basis for research into custom acceleration [12].

Private and public clouds and research testbeds are increasingly being fitted with such hardware. Such testbeds include FABRIC [13] and DETER [14]. Testbeds that have not yet been fitted with this hardware are being used for programmable networking research using soft-switches [15].

Unlike non-programmable networks, programmable networks can involve in-network state and dynamic changes brought about by in-network programs. Thus a key challenge when operating programmable networks involves reasoning about the combination of in-network programs that are simultaneously executing across heterogeneous hardware targets.

A reasoning approach that is being explored for programmable networking builds on the *dataplane disaggregation* idea [16] which starts with a monolithic in-network program representing the computation being done across the network, splitting it into smaller parts, and distributing its logic and state among different devices in a programmable network.

This approach is implemented in the Flightplan system,[1] and it is a stepping stone towards a more general resource-based reasoning approach for programmable networking. In the current approach, the *Flightplanner* tool automates reasoning about heterogeneous resources, in-network programs, and user objectives (such as trading-off latency for power saving). Flightplanner allocates program splits to devices in a programmable network.

But in its current form, Flightplanner is difficult to generalise and extend: **(1)** It introduces a new, custom formalism that lacks mathematically-founded semantics. An example of this formalism is shown in Rule 1. This formalism encodes different types of information, including an abstraction of the in-network program and the capabilities of heterogeneous network hardware. **(2)** It relies on a custom and complex reasoning engine for this formalism.

This paper contributes Lightplanner (§IV): a reasoning tool for a well-founded resource- and programming-model for distributed and heterogeneous systems, and that is compatible with Flightplanner. Lightplanner **(1)** Embeds Flightplan's rules in Prolog, giving them semantics in a well-understood formalism. Listing 1 shows a snippet of the adaptation of Rule 1 into Prolog. **(2)** Implements the planner in Prolog itself, relying on Prolog's inference and search to find and check plans. Fig. 1 shows how the workflows of the two planners differ.

[1]Flightplan is open-sourced at https://flightplan.cis.upenn.edu

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

Rule 1. Using Flightplan's formalism, this rule captures the performance profile when the **header_compress**() function is executed on a CPU[3] at a specific range of workload characteristics (packet sizes and throughputs). The $[\cdots]$-notation captures the effect that **header_compress**() has on the abstract state that is maintained by Flightplanner. The $[\cdots]$-notation describes how variables may be updated every time the rule is used in a proof, or, using the 'once' qualifier, updated once throughout the whole plan.

```
% CPU header_compress 0.02
profile(_, header_compress, prop_CPU, St0, St1) :-
...
  stateVal(bound_InputRate, St0, InputRate),
       stateVal(bound_InputRate, St1, InputRate2),
  stateVal(bound_PacketSize, St0, PacketSize),
       stateVal(bound_Latency, St0, Latency),
  stateVal(bound_Latency, St1, Latency2), InputRate2
        is InputRate * 0.975096277278562,
  InputRate =< 200000000, PacketSize >= 1000,
       Latency2 is Latency + 0.00740.
```

Listing 1. Prolog embedding of the performance profile for `header_compress()` running on a CPU. Lightplanner uses this rule to explore allocations of this function. Compared to the custom formalism used in Flightplanner (cf Rule 1), the Prolog version of this rule uses a more widely-recognised notation and widely-implemented inference system.

Lightplanner is $10\times$ smaller than Flightplanner's C# implementation but can be used as a drop-in replacement for it. Both systems accept exactly the same inputs and implement the same behaviour. Lightplanner's implementation is smaller and simpler because of the reliance on Prolog to provide the formalism and search. In comparison, Flightplanner implemented those facilities from scratch. Lightplanner's smaller and simpler implementation makes it more amenable for formal analysis in future work.

Lightplanner was evaluated (§V) on the Flightplan benchmark. Lightplanner is being developed as part of research on managing programmable networks, and is initially being used on a local programmable testbed network at Illinois Tech.

II. Background

The P4 [9] language is used for in-network computing on various hardware targets. Setting up a running example in this paper, Listing 2 shows P4 snippet taken from the Flightplan paper [16, §2].[4] The code snippet is executed before the packet is forwarded by the network element and behaves as follows: The look-up to table `egress_compression` on line 3 determines whether a packet should be compressed by inspecting which network port it is being egressed to. That port information is stored in the `meta.egress_spec` variable. As a side-effect, the look-up changes the value of the `compressed_link` variable. Based on this variable's value, the code might branch on line 4.

[3]Specifically an 8-core Intel Xeon 2450.

[4]The full program can be found at: https://github.com/eniac/Flightplan/blob/master/Wharf/splits/ALV_Complete/ALV_Complete.p4#L243

```
1  flyto(Compress);
2  // If heading out on a multiplexed link, then
      compress header.
3  egress_compression.apply(meta.egress_spec,
      compressed_link);
4  if (compressed_link == 1) {
5     header_compress(forward);
6     if (forward == 0) {
7        drop();
8        return;
9     }
10 }
```

Listing 2.
Snippet of P4 code. Green highlight indicates resource-related syntax that is analysed for dataplane disaggregation to ensure that it is allocated to hardware targets that are able to execute it. Orange highlight indicates annotation used in Flightplan to *segment* the code—setting up potential points where to split the program.

The `header_compress()` function, featured in Rule 1, is called on line 5. It sets the `forward` variable to indicate whether `drop()` should be called. This is called to drop the packet to prevent duplication since `header_compress()` returns the original packet as well as a compressed equivalent.

III. Experiment Planning for Heterogeneous Programmable Networks

Building on the approach developed in Flightplan, we model the programmable network as a single P4 program, then use a toolchain like the one shown on Fig. 1 to analyse different ways of splitting up the program and mapping it to the available hardware based on knowledge of the network's topology, the hardware's capabilities, and program-level requirements expressed by the network operator. Program segments A-E in Fig. 1 can end up being separate, but cooperating programs, and executing on different hardware.

Subprograms—called *segments*—are defined declaratively and named. Listing 2 shows a segment called `Compress`. The Flightplan analyser generates a rule in its formalism for each code-path through a segment. In Listing 2 there are three code paths. They are formed by the evaluation of "`compressed_link == 1`" and "`forward == 1`". The maximal sequence of resources used in this segment are: egress_compression; header_compress; drop.

Resources are described formally using *performance profiles* which capture effects of using a specific resource on a specific piece of hardware. Rule 1 is an example performance profile rule for the `header_compress()` function we saw on line 5 of Listing 2.

Flightplanner uses a variety of rule-encoded information to produce *plans*. Forming a plan involves navigating the network topology—starting at a user-indicated device—and finding an allocation of segments to devices on the network. The planner uses rules to compose a proof that the chosen target is sufficiently resourced to execute a given segment.

IV. Lightplanner

Lightplanner emulates the behaviour of Flightplanner—it accepts the same inputs and provides the same outputs—

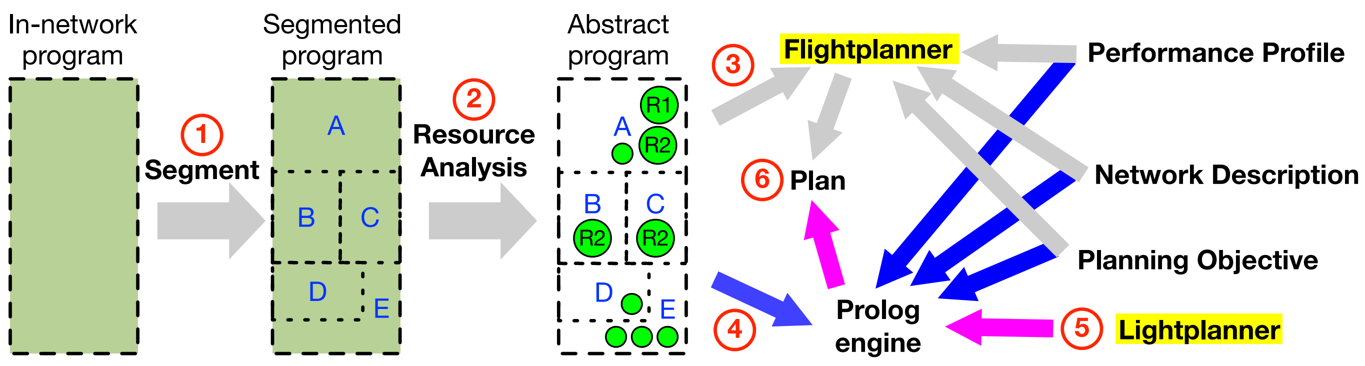

Figure 1. Plan-producing workflow used in Flightplan (grey arrows) and those added by Lightplanner (blue and purple arrows). The two planners are shown highlighted. Blue arrows indicate automatic conversion of input data into Prolog programs, and the purple arrows show the path to producing plans using Lightplanner by relying on a third-party Prolog engine rather than an ad hoc reasoning tool like Flightplanner. *Step-by-step workflow*: ① An in-network program is *segmented* using Flightplan's annotations. Segments are named, and this illustration shows segments that are named A-E. Segments represent the finest granularity at which the program can be split. ② The segmented program is analysed to determine the resources used by each segment. Resources include tables, types of memories, and external functions. In this drawing, resources are illustrated using green circles; different segments can use different quantities and types of resources. In this drawing we see that segments B and C use the same type of resource (R2) while A uses an additional type of resource (R1). Therefore A must be allocated to hardware that provides both resources, but B and C can be allocated to different, possibly weaker, hardware. This stage produces an *abstract program* that only references the kinds of resources used by each segment. ③ In Flightplan, all the inputs converge into Flightplanner which automates reasoning using rules in an ad hoc formalism. ④ This paper describes a deviation of Flightplan's workflow to (i) convert (blue arrows) the inputs into Prolog and (ii) converge them into a Prolog engine where they are joined by the ⑤ Prolog-implemented Lightplanner tool. ⑥ This new approach produces plans by relying on Prolog proof-search instead of Flightplanner's approach of using a custom proof engine and rule definition.

```
tgt(DeviceName, compress_Seg5, Target, St0, St3B) :-
  tgt(DeviceName, [compress_Seg5,1], Target, St0,
      St1A), maxStateList(St1A, St2A, St2B),
  tgt(DeviceName, [compress_Seg5,2], Target, St0,
      St2A), maxStateList(St2B, St3A, St3B),
  tgt(DeviceName, [compress_Seg5,3], Target, St0,
      St3A).
```

Listing 3. Abstract program rule for the segment from Listing 2. This rule picks the maximum-cost code-path (using the chained `maxStateList/3`) to ensure that the planning will allocate it conservatively—that the target hardware can satisfy the most demanding of code-paths. `compress_Seg5` is a Prolog-mapped name for Compress.

but relies on Prolog's proof search instead of Flightplanner's custom reasoning engine (cf Fig. 1). Lightplanner embeds a generalisation of Flightplan's formalism in Prolog. This generalisation is versatile enough to describe device capabilities, segments, and code paths. Continuing with our running example, Listing 3 shows the embedding of the entire Compress segment, decomposing it into its code-paths. The conversion of these rules is automated using simple and reusable scripts. Other information is encoded directly in Prolog: the network topology is encoded as a graph, device information as a relation, and objective function as a list.

V. Evaluation

Lightplanner was evaluated in two ways: **(1)** The size and complexity of Lightplanner's implementation was compared to that of Flightplanner. **(2)** The comparative effectiveness of both systems was evaluated using the Flightplan benchmark in terms of (a) solution quality and (b) time taken to find a solution.

a) Implementation size and complexity: Lightplanner's implementation (354 lines of Prolog in a single file) is 10× smaller than Flightplanner's (3584 lines of C# spread across 11 files). Lightplanner is also simpler than Flightplanner. The C# implementation used the language's type system to form sophisticated representations of different types of information—including proofs, plans, programs, and hardware—and used nested coroutines to implement lazy search in a strictly-evaluated language. In the Prolog implementation, tuples and relations are used to represent the required information, and Prolog natively supports search.

b) Effectiveness: This evaluation uses the benchmark suite that was publicly released as part of Flightplan.[5] It consists of 20 variants of P4 programs that were processed by Flightplan's analyser, together with the other inputs to the Flightplanner, including the network and device description. Both planners were configured to use the same objective function: minimize latency, cost, and power, in that priority.

Table I shows the results for both planners. The symbol ⊥ indicates resource-exhaustion, meaning that: *either* a memory limit was reached, and an out-of-memory handler was used to kill the process (for Flightplanner) or the local or global stack sizes were exceeded (for Lightplanner), *or* a 100-second timeout expired. Lightplanner's default mode—column LP in Table I—produces the best results: the quality of the results are either as good as those of Flightplanner, or exceed them. Numbers highlighted in yellow indicate that the quality of the solution found by that column's tool was better than those of the other tool's corresponding mode. Lightplanner's greedy mode—column LP(G)—produces less good solutions than Flightplanner's greedy mode, FP(G). In most cases there is little time saving between LP(G) and LP, making the heuristic less useful on this benchmark. Although FP(G) was able to

[5]https://github.com/eniac/Flightplan/blob/master/flightplanner/examples/

Table I
Performance comparison between Flightplanner (FP) and Lightplanner (LP).
'Prolog #Lines' shows the size of the Prolog program generated for each benchmark.

Bench.	**Time (milliseconds)**				**Prolog**	**Bench.**	**Time (milliseconds)**				**Prolog**
#Prog.	FP	FP(G)	LP	LP(G)	**#Lines**	**#Prog.**	FP	FP(G)	LP	LP(G)	**#Lines**
1	60	230	140	140	41	11	660	80	140	130	39
2	130	110	190	170	43	12	60	60	130	130	33
3	110	100	150	160	43	13	60	60	140	130	33
4	80	60	140	140	41	14	80	70	140	130	36
5	70	70	270	270	557	15	830	910	30K	30K	47664
6	120	90	240	240	136	16	$\perp$	670	230	760	604
7	60	70	270	270	557	17	500	390	5K	5K	12369
8	110	100	230	240	136	18	400	400	5K	5K	12369
9	50	60	140	140	33	19	$\perp$	130	4.5K	300	162
10	50	60	130	140	33	20	$\perp$	1.6K	15K	280	166

find a solution to program 20, the solution found by LP was superior—it used less equipment, requiring less power and less cost. In program 15 there appears to be a high time-cost to load the problem, resulting in little difference between LP and LP(G). Future work will investigate better heuristics, and better tuning for the greedy heuristic.

VI. Future Work

Lightplanner has feature parity with Flightplanner yet is smaller and simpler, making it better suited for additional research, and for others to understand and extend. One item for future work involves pre-processing the output from Flightplan's analyser to reduce the size of the abstract program. Even simple P4 programs—such as programs 15, 17, and 18 in Table I—can yield large abstract programs. A trade-off in abstraction accuracy could be used to simplify the resulting abstract program. By simplifying abstract programs we could improve the planner's scalability to handle more complex distributed P4 programs.

Another item for future work involves adapting Lightplanner to schedule workloads in our local testbed—and later in a large scale distributed testbed—for a variety of applications and programmable network hardware.

Acknowledgement. I thank the TEPN reviewers for their feedback. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-19-C-0106. 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] D. R. K. Ports and J. Nelson, "When Should The Network Be The Computer?" in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 209–215. [Online]. Available: https://doi.org/10.1145/3317550.3321439

[2] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 51–66.

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

[4] Netcope Technologies, "Netcope P4," https://www.netcope.com/en/products/netcopep4, Jun. 2017.

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

[6] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN," *SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, Aug. 2013.

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

[8] Broadcom Inc, "Broadcom Trident 3," Jun. 2017.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[10] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 385–398.

[11] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *Demo presented at ACM SIGCOMM*, 2015.

[12] D. Sanvito, G. Siracusano, and R. Bifulco, "Can the Network Be the AI Accelerator?" in *Proceedings of the 2018 Morning Workshop on In-Network Computing*, ser. NetCompute '18. New York, NY, USA: ACM, 2018, pp. 20–25.

[13] I. Baldin, A. Nikolich, J. Griffioen, I. I. S. Monga, K.-C. Wang, T. Lehman, and P. Ruth, "FABRIC: A National-Scale Programmable Experimental Network Infrastructure," *IEEE Internet Computing*, vol. 23, no. 6, pp. 38–47, 2019.

[14] J. Wroclawski, T. Benzel, J. Blythe, T. Faber, A. Hussain, J. Mirkovic, and S. Schwab, "DETERLab and the DETER Project," in *The GENI Book*. Springer, 2016, pp. 35–62.

[15] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The Design and Operation of CloudLab," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 1–14. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/duplyakin

[16] N. Sultana, J. Sonchack, H. Giesen, I. Pedisich, Z. Han, N. Shyamkumar, S. Burad, A. DeHon, and B. T. Loo, "Flightplan: Dataplane Disaggregation and Placement for P4 Programs," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 571–592. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/sultana