


Automatic Generation of Network Function Accelerators Using Component-Based Synthesis

Francisco Pereira   Gonçalo Matos  Hugo Sadok  Daehyeok Kim 
Ruben Martins  Justine Sherry  Fernando M. V. Ramos  Luis Pedrosa 

 INESC-ID, Instituto Superior Técnico, University of Lisbon

 Carnegie Mellon University

 Microsoft

Abstract

Designing networked systems that take best advantage of heterogeneous dataplanes – *e.g.*, dividing packet processing across both a PISA switch and an x86 CPU – can improve performance, efficiency, and resource consumption. However, programming for multiple hardware targets remains challenging because developers must learn platform-specific languages and skills. While some ‘write-once, run-anywhere’ compilers exist, they are unable to consider a range of implementation options to tune the NF to meet performance objectives. In this short paper, we explore preliminary ideas towards a compiler that explores a large search space of different mappings of functionality to hardware. This exploration can be tuned for a programmer-specified objective, such as minimizing memory consumption or maximizing network throughput. Our initial prototype, SyNAPSE, is based on a methodology called component-based synthesis and supports deployments across x86 and Tofino platforms. Relative to a baseline compiler which only generates one deployment decision, SyNAPSE uncovers thousands of deployment options – including a deployment which reduces the amount of controller traffic by an order of magnitude, and another deployment which halves memory usage.

Keywords

In-network compute, Network function virtualization, Programming abstraction

1 Introduction

In the pursuit of upgradeability, customizability, and innovation, the networking community has embraced programmable dataplanes to implement network functions (NFs), including advanced features like deep packet inspection, filtering, address translation and WAN optimization. Which

programmable hardware is best suited to implement these capabilities remains an active area of debate: will the winner be Protocol Independent Switch Architecture (PISA) switches [2, 3, 7, 8]? Network Processing Units [20]? FPGAs [32]? Or x86 software [9, 22, 24, 25]?

The challenging truth is that all of these platforms have their strengths. For example, while PISA switches excel at match-action operations on packet headers [18], traditional x86 software wins for parsing regular expressions over payloads [32]. Hybrid dataplane designs [10, 23, 27, 31, 32] which use an ensemble of platforms, each platform performing tasks it is best suited for, are proving capable of higher throughput, lower latency, better energy efficiency, lower cost, *etc.* than single-platform approaches. For example, traditional reactive SDN partitions the dataplane between packet forwarding (performed entirely on the switch), and flow establishment (with a controller), and some intrusion detection systems partition the dataplane between exact match search (using a SmartNIC) and regular expression parsing (on a CPU) [5, 32].

Unfortunately, hybrid designs are extremely challenging to develop. Each programmable platform comes with its own programming language (*e.g.*, P4, Verilog, or C), hardware features (Is there SRAM, DRAM, or both? Is there a cache hierarchy?), and debugging challenges. Developing high-performance NFs for any single platform is already widely considered to be challenging, and asking developers to learn an elite skillset for multiple such platforms is a very tall order. What developers really need is the ability to ‘write once, and run anywhere’ with code not only portable across different platforms, but *partitionable*, with dataplane codepaths automatically provisioned on the hardware best-suited for that task.

Groundbreaking forays into ‘write once, run anywhere’ models have shown that it is possible to build compilers and schedulers that can correctly partition code across multiple dataplane platforms [6, 10, 23, 27, 31]. Nonetheless, existing approaches fall short of delivering on the core premise of hybrid dataplanes: the ability to use each platform for what it is best suited with regard to a particular performance or resource objective such as latency or memory utilization, respectively. At a high level, existing approaches focus primarily on the problem of *translation*, reasoning about performance or resource utilization only – if at all – with regard to one,

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SOSR 22, October 19–20, 2022, Virtual Event, USA

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9892-3/22/10.

<https://doi.org/10.1145/3563647.3563656>

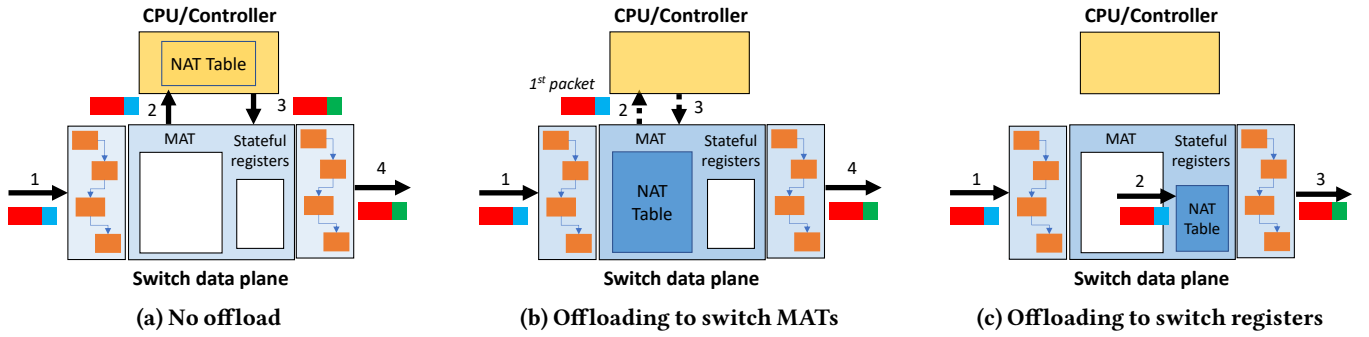


Figure 1: Three NAT implementations: a) to save switch resources, all packets go to the controller and state is kept in CPU memory; b) to improve throughput while avoiding limited register memory, state is kept in the switch MAT. Only the control-plane can update the MAT, so the first packet of a flow goes to the controller. c) to maximize throughput, state can be kept in registers, which can be updated from the data-plane without controller intervention.

Algorithm 1 Simplified NAT pseudo-code. *ports* maps flows to public ports, and *flows* has the reverse mapping.

```

1: function PROCESS(pkt, port)
2:   if port = WAN then
3:     if flows.contains(pkt.d_port) then
4:       pkt.d_ip ← flows[pkt.d_port].priv_ip
5:       pkt.d_port ← flows[pkt.d_port].priv_port
6:       forward(LAN)
7:     else
8:       drop()
9:   else
10:    flow ← {pkt.s_ip, pkt.d_ip, pkt.s_port, pkt.d_port}
11:    if ports.contains(flow) then
12:      public_port ← ports[flow]
13:    else
14:      public_port ← allocate_port()
15:      ports[flow] ← public_port
16:      flows[public_port] ← {pkt.s_ip, pkt.s_port}
17:    pkt.s_ip ← PUBLIC_IP
18:    pkt.s_port ← public_port
19:    forward(WAN)

```

hard-coded objective.

In this paper, we discuss early design considerations for a cross-platform compiler which tunes deployment of a given network function to achieve performance or resource objectives. For example, a NAT designed for maximum throughput might store all port mappings on a switch dataplane, but a NAT designed to conserve on-switch memory might only store state for elephant flows on the switch dataplane, and keep mice on an external controller. Our prototype compiler, SyNAPSE, reasons about such trade-offs and generates fine-tuned deployments based on component-based synthesis [13].

The key insight behind SyNAPSE is that much of the performance and resource tuning done by elite developers is

invested into the implementation of common, reusable algorithms and data structures. Typical developers then re-use these algorithms and data structures, choosing implementations that best suit their performance objectives (e.g., using a hash table with separate chaining for memory density, or cuckoo hashing [21] to put tight bounds on lookup times).

SyNAPSE provides a library of standard abstract data types (e.g. map, vector) and algorithms with multiple backing implementations; we broadly call these ‘components’. For example, in software SyNAPSE might provide a cuckoo hash, a binary search tree, and a hash table with separate chaining all to implement map, and for PISA switches SyNAPSE might use registers as well as a TCAM or SRAM-based tables to implement the same component. Hence, the same component can be implemented on multiple platforms, and multiple ways on each platform. Importantly, component implementations can be written once by developers with platform-specific expertise, and then be reused later by developers who need not worry about hand tuning the internals of each component.

NF developers need only focus on the control flow and networking logic of their application, programming against the abstract SyNAPSE component APIs. By separating the component implementations from NF core logic, we can use exhaustive symbolic execution techniques [30] to build a sound and complete representation of NF functionality. At compile-time, the SyNAPSE compiler takes in this representation, a profile of the available hardware platforms and their resources, and a performance or resource objective to optimize for. It then searches over the space of *all possible* deployments to find a suitable mapping of components to hardware.

In comparisons relative to state-of-the-art Gallium [31], our initial prototype can generate a deployment of a Network Address Translator over a PISA switch and x86 controller that matches their resource utilization and throughput. However, because SyNAPSE can be asked to optimize for different performance metrics, it is also able to find a deployment which provides higher throughput, reducing the amount of traffic

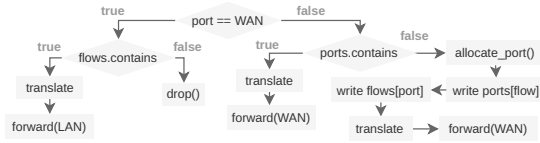


Figure 2: NAT model.

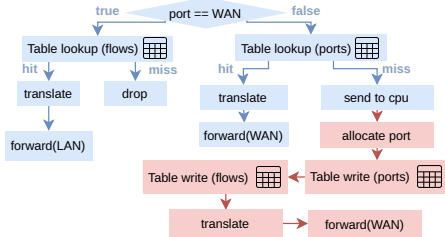


Figure 3: Switch and controller NAT logic optimized for resource utilization. nodes use switch tables.

to the controller by 3 orders of magnitude (at a cost of switch memory resources), and a deployment which optimizes to minimize switch resources, halving the amount of SRAM used (at the cost of higher CPU load on the controller).

2 Motivation

To understand the process of offloading an NF to a network accelerator, consider a Tofino-based programmable switch [8] and the simplified NAT in Algorithm 1. New flows from the LAN to the WAN interface are remapped to use a unique external port (Line 14). All of the information needed to perform address translation for future packets in this flow is then stored in two maps (for LAN to WAN traffic: Line 15; and vice-versa: Line 16). As traffic comes back from the WAN, it either matches a known flow and is translated/forwarded (Line 6), or is dropped (Line 8).

There are many ways of offloading this NF to the switch, three of which we outline in Figure 1. Each of them showcases a specific switch resource being allocated to store the NAT’s translation table. Figures 3 to 5 complement these by depicting three distinct offloading implementations that make use of this resource utilization flexibility to optimize for specific goals. We call these offloading solutions *execution plans*. Each execution plan’s node represents some logic implemented either on the switch (if the node is blue) or on the controller (in case it is red).

Each different offloading approach represents a particular trade-off between performance and resource-usage. Figure 1a illustrates a trivial solution that steers all packets to the controller. Although the NF is no longer accelerated, it is a valid option, for instance, if the goal is to save the switch’s resources to be used by other applications.

Other solutions (e.g., Figures 1b and 3) would implement the maps *flows* and *ports* as P4 tables to be kept in the Match-Action Tables (MATs). As updating a map corresponds to

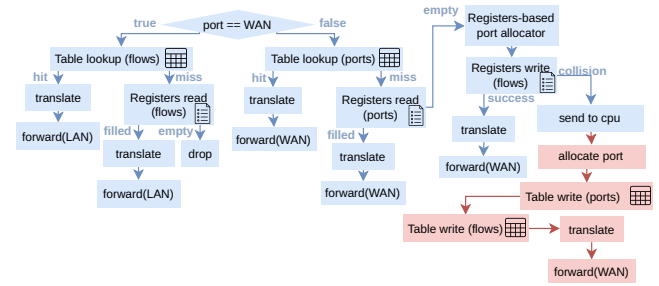


Figure 4: Switch and controller NAT logic optimized for CPU load. nodes use switch tables, use registers.

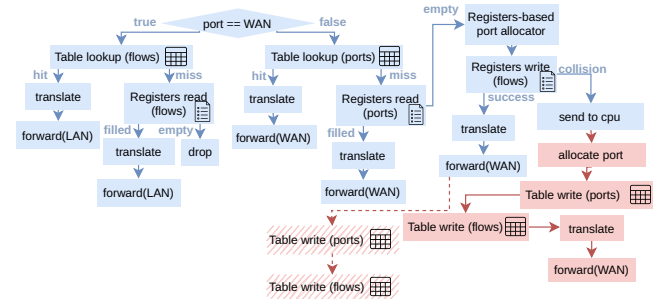


Figure 5: Switch and controller NAT logic optimized for throughput. nodes use switch tables, use registers. Hashed nodes represent async operations.

adding a new entry to the MAT, and as MATs can be updated from the controller only, the first packet of a flow needs to be steered to the controller, triggering the installation of a new flow rule in the switch. Subsequent packets can be processed entirely in the switch data plane thereafter. This accelerated version (we call it **min-resources**) improves the performance by keeping most traffic in the data plane, at the cost of switch memory. This is similar to the accelerated NAT generated by Gallium [31] that also requires CPU intervention to install new flows in the switch.

For maximum performance, a Tofino expert would try to maintain packet processing *entirely* in the switch data plane (Figure 1c). This would require the use of data structures on the Tofino that allow write operations by the data plane itself, e.g., P4 registers. These are hard to use, with restrictions like only allowing a single read-modify-write operation per packet, but can be used to perform write operations to state stored on the data plane. The expert could envision many different solutions to this problem. For instance, one solution would be to store a limited number of flows in these registers using a simple hashing mechanism (Figure 4). As register space fills up, flows will cause hash collisions, at which point the new flow can be directed to the controller to populate an overflow space in tables. As most traffic stays in the switch, we call it the **min-cpu-load** solution.

Another solution would take this a step further and asynchronously move flows from the limited register space to the larger MAT tables (Figure 5), trading off some additional CPU load for this process for higher network throughput (**max-throughput**). We could even take advantage of more exotic algorithms and data structures on the data plane already devised by previous work, such as sketches [16, 29], bloom filters [18], or cuckoo hashes [18]. All of these, or even some combination of them, could potentially be used to form an even better performing offloading version of a NAT.

One key takeaway is that there is a plethora of different ways to offload this single NF to the switch, all of them with different trade-offs and payoffs. Although part of the challenge is planning how to partition logic and data between the accelerator and the controller, it actually goes deeper: *even for the same partitioning one could devise different algorithms to implement the same functionality*.

While offloading NFs to network accelerators in these various manners can improve their performance and free up computational resources on commodity hardware, it is not trivial how to do it since it requires deep expertise and familiarity with the accelerator’s idiosyncrasies. Moreover, each time the NF implementation changes, its offloaded solution might need to be completely restructured. Automating this task can potentially bring all its benefits whilst taking the labor off the shoulders of the NF developers.

State-of-the-art and SyNAPSE: Recent work tackled the challenge of automating NF offloading by either requiring NFs to be developed in a high-level programming language specifically designed for this purpose [6, 10], using machine learning to infer porting strategies [23], or by partitioning accelerated code (P4) into multiple P4 targets [27]. Closer to us, Gallium [31] starts from an NF written as a Click module [14] and builds a model for the NF which it then manipulates and partitions to try to maximize the amount of operations offloaded to the accelerator, which for their NAT results in something like Figures 1b and 3. This approach is akin to a compiler, transforming the NF using to a set of rules to find a single solution.

SyNAPSE pushes on the state-of-the-art in three ways. First, it uses as input NF code written in a general-purpose language (*e.g.*, C). As a result, the developer does not need elite skills in different low-level languages and in-depth knowledge of the target hardware intricacies to program the accelerated code. Second, it leverages program synthesis that navigates the vast space of possible offloading solutions to search for the configuration that best meets the user’s goal – a vast space that is mostly overlooked by previous systems. Within this space would be Gallium’s solution and with the right configuration SyNAPSE could find it, but the power of our approach is in the reasoning about this space, rather than the ability to come up with any one given solution. Third, our component-based

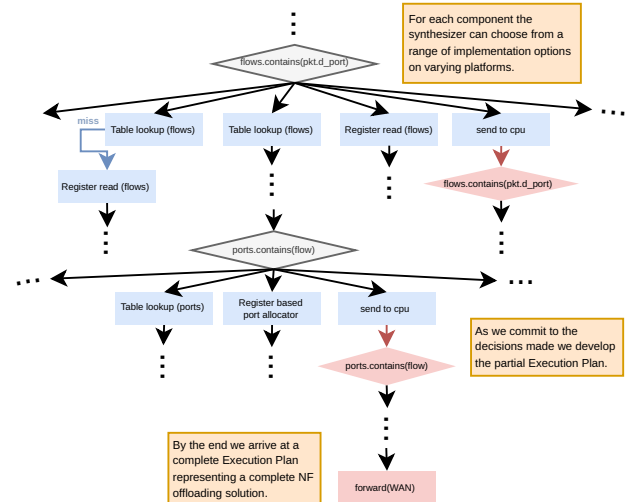


Figure 6: Search building the execution plans. approach [13] separates the development of the core NF logic from code acceleration.

3 Component-based NF Synthesis

In this section, we discuss the SyNAPSE design and the different roles different kinds of developers take.

Development: We envision two kinds of developers: accelerator developers and NF developers. Accelerator developers are platform experts. They implement reusable components respecting the same abstract interface, like implementing a lookup table using a P4 table for PISA switches or a hash table for x86 CPUs, and expose them as C APIs. These components are made available to NF developers, *e.g.*, via open source libraries. NF developers write C-like NF code (with some constraints on loops and pointer arithmetic to facilitate analysis [30]) that use the components in an NF implementation.

Deploying NF Code: Using our NAT running example, SyNAPSE starts from an NF implementation provided by an NF developer (Section 2) and performs symbolic analysis [4] to extract a sound and complete representation of its functionality. This follows the approach developed in Vigor [30], whereby data structures and other complex library code are separated out from the otherwise stateless NF core logic. Without the complexity of data structures and algorithms, the NF code is now stateless. Hence it is amenable to exhaustive symbolic execution, resulting in a tree representing all possible code paths. Subjecting our NAT to these techniques outputs the model depicted in Figure 2. This tree represents NF functionality in an abstract manner, tracking how input packets and previous state are transformed into output packets and new state.

The next step explores ways to convert our tree of code paths into a concrete execution plan (or EP, as in Figures 3 to 5) which maps NF functionality to platform-specific functionality. The search process gradually builds partial EPs by replacing nodes in the functional representation with components

that implement the same functionality on a specific platform.

Figure 6 shows a portion of the search space that SyNAPSE navigates when offloading the NAT to a programmable switch, specifically using the P4 language. During this search, it detects that an access to the map flows could be implemented as either a simple table lookup, a direct register lookup, or even a more complex mechanism that uses both tables and registers. Another alternative is sending the packet to another device, *e.g.*, the software controller, another accelerator, or even a commodity middlebox server. This allows the search process to explore solutions that use multiple heterogeneous accelerators that distribute functionality across the network.

Starting with the functional tree described earlier, the search progressively tries replacing nodes one by one to emit new EP nodes. At each step, SyNAPSE picks which tree node to convert next, taking into account data-dependencies and read/write semantics, and then selects which component to swap in, out of the many possible implementation candidates. As the tree is explored, each partial EP is scored using a search heuristic, guiding the search process towards finding “better” solutions, according to some user-defined measure of goodness. The search is complete when all functional nodes have been converted to EP nodes. Crucially, all complete EPs that can result from search will be valid implementations that vary in performance *but not functionality*.

Figures 3 to 5 correspond to valid EPs of Figure 2 but with different trade-offs, all reachable via different heuristics. For Figure 3, SyNAPSE chose to implement the port allocation logic on the controller, whereas for Figures 4 and 5 it decided to optimistically implement this stateful logic using registers on the switch, with the latter having additional logic implemented to allow for asynchronous migration of data between the registers and tables. Crucially, all these choices were motivated by their specific heuristic.

Heuristics: In the above discussion, we did not describe how SyNAPSE decides *which* of the many possible component implementations to choose in converting the functional tree into an EP. Deciding which component implementation to choose at each step is the job of a heuristic. The design of good heuristics remains an open problem for the SyNAPSE project, but we envision pre-programming SyNAPSE with several heuristics, and at NF deployment time an operator simply choosing the heuristic that best meets their needs. For example, in a datacenter with online data-intensive applications [28] and tight deadlines, an operator might choose to run a heuristic designed to minimize latency; in a wide area network setting with large traffic flows an operator might prefer a heuristic which maximizes throughput.

We define heuristic as a function that given a pair of partial EPs returns the *better* of the two (*i.e.* a comparator), for some definition of *better*. For example, a simple heuristic that tries

to optimize for throughput, when given both options, might choose an implementation that does not detour traffic to the controller over one that does. The search process uses the heuristic to sort and decide which partial EP to make progress on during the next iteration, as it is deemed the *best* candidate so far. That partial EP will potentially generate multiple others, as SyNAPSE will probably find that there are multiple ways of offloading the current component. The process then repeats, continuously using the heuristic to guide search.

Our work developing heuristics is still at an early stage, and so far we have explored a few simple heuristics such as minimizing the number of EP nodes that run on the CPU. However, we hypothesize that more elaborate heuristics have potentials to outperform the simple heuristics we have tried. In the examples above about minimizing latency or maximizing throughput, we might incorporate a performance prediction model into our heuristics. Exploring the best heuristics for SyNAPSE across a wide range of platforms is our future work.

4 Evaluation

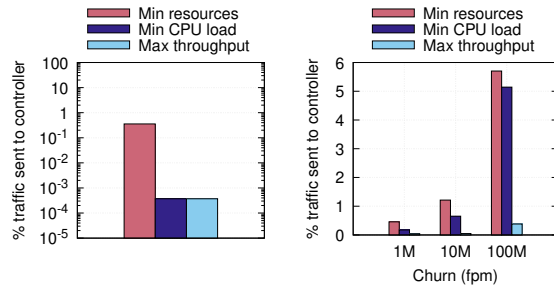
In this section, we conduct experiments to understand the impact that different performance targets have on different aspects of the systems that SyNAPSE generates. We implement a prototype of SyNAPSE and run it using the same running example of a NAT described in Section 2. We implement the NAT in C with the Vigor framework [30].

As our work on heuristics is still early stage, we manually explore the search space. We found three distinct solutions targeting different performance objectives: CPU load, resource utilization, and throughput; with solutions illustrated in Figures 3 to 5. We refer to these solutions as `min-resources`, `min-cpu-load`, and `max-throughput` respectively. We then evaluate the performance of the different solutions by looking at three different factors: the fraction of packets that the switch sends to the controller, the CPU load on the controller, and the switch resource utilization.

4.1 Experimental setup

Simulator: We implement a simulator that replicates the switch functionality and the controller in software. Because we have full control over the simulator we can retrieve fine grained statistics about the behavior of the switch and how it interacts with the controller. For instance, we can track the number of times that the controller writes to the data structures inside the switch as well as the occupancy of these data structures. Using the simulator also lets us more easily generate synthetic traffic with variable packet sizes and packet inter-arrival times.

Time is simulated and packets are transmitted as if they were subjected to a link rate of 100 Gbps. The simulated switch data structures mimic the same capacities as the ones on the



(a) Empirical.

(b) Synthetic.

Figure 7: Fraction of traffic sent to controller under empirical (a) and synthetic traffic with variable churn (b).

real switch. Moreover, the simulation uses the same hash function to calculate register indexes as on the switch, which is crucial to detect collisions. Finally, it takes $300\mu\text{s}$ to complete write operations to the tables [31].

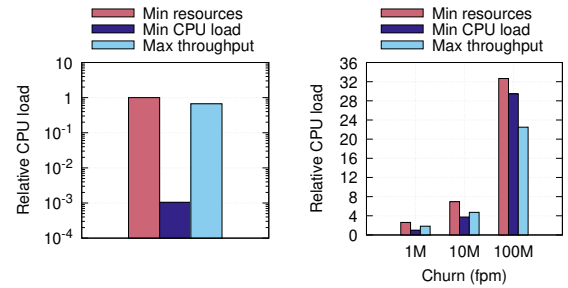
These simulations are, however, not meant for fine-grained performance prediction, but for providing insights on relative performance between different offloading solutions optimized for different goals. We envision that operators may use them to test different performance objectives before they even commit to buying new hardware.

Workloads: We use two types of traffic: empirical traces obtained from a university datacenter [1], and synthetic ones. Both contain 42k flows in total and 117 elephant flows responsible for 80% of the total volume of traffic. The packet sizes follow a bimodal distribution in which the majority is less than 200B or greater than 1400B. The synthetic traffic has the same characteristics as the empirical one but modified to achieve specific values of flow churn, which we define as the number of different flows that replace others during a given period. Traces with higher churn have to handle new flows more often. We found that churn is a particularly important metric for many network functions generated using SyNAPSE as packets from new flows are often more costly than packets from existing ones.

4.2 Findings

Fraction of traffic sent to the controller: Figure 7a shows how different performance objectives impact the fraction of packets sent to the controller. Under empirical traces, the *min-resources* configuration causes the amount of traffic sent to the controller to increase by three orders of magnitude compared to the other configurations. This is expected as sending more traffic to the controller allows SyNAPSE to use fewer switch resources.

With no recirculation the Tofino processes packets at line rate independently of packet sizes. Performance not only deteriorates but becomes harder to predict when packets are sent to the controller. That motivated us to use the relative number of packets sent to the controller as an evaluation metric. This



(a) Empirical.

(b) Synthetic.

Figure 8: Relative CPU load under empirical (a) and synthetic traffic with variable churn (b).

metric is optimistic in the sense that it does not consider the time it takes for a packet to be sent to the CPU and trigger a table write operation in the switch to completion. Nevertheless it provides an important insight on the NF’s performance on the switch: the more packets are sent to the controller, the worse the performance.

We also use synthetic traces to show that the fraction of packets sent to the controller correlates with the degree of churn, measured in flows per minute (fpm) (Figure 7b). While this is true for all configurations, *max-throughput* is the least sensitive to churn. This is because, unlike *min-resources*, which requires the first packet of every flow to be sent to the controller, *max-throughput* takes advantage of registers, and thus processes a larger fraction of traffic entirely in the switch. Moreover, unlike *min-cpu-load*, it relies on the controller to asynchronously move data from the registers to the tables. This decreases the number of flows that need to be sent to the controller, which happens every time there is a collision on the registers. The trade-off to this solution is that it increases the load on the CPU in the common case, as we will see next.

CPU load on the controller: The CPU load on the controller is influenced not only by the amount of traffic that it receives but also by the number of asynchronous operations that the application imposes. In this context, asynchronous operations are those that are conducted without the need to forward traffic to the controller. For example, the *max-throughput* configuration uses asynchronous operations to move flows from registers to tables, without sending the first packet of every flow to the controller. Instead, a digest is sent to the controller, which eventually acts upon it and migrates the new flow’s data from the registers to the tables. Even though this is able to reduce the amount of traffic sent to the controller, it still imposes load on the CPU. Figure 8a demonstrates this by showing the relative CPU load on the controller for the three different configurations under the empirical traces.

Surprisingly, however, when we artificially increase the level of churn to 100M fpm using the synthetic traffic (Figure 8b), the *min-cpu-load* configuration starts to impose an even larger CPU load on the controller than *max-throughput*.

This is due to the fact that, under such extreme churn, the register space fills up fast and both solutions end up sending either a digest or a full packet to the controller for every new flow. The way `max-throughput` processes the digest, however, induces slightly less load than how `min-cpu-load` processes packets. Whereas the latter receives the packet, updates the switch tables, and finally forwards it, the former needs only to receive the notification (digest) and update the tables.

Switch resources: Finally, we look at the amount of switch resources that each configuration needs. Table 1 shows the resource utilization reported by the Tofino compiler as a percentage of each resource type used when targeting a 32-port 100Gbps Tofino switch (Wedge 100B-32X). It shows how SyNAPSE can vastly reduce switch resource utilization when this is set as the main performance objective (`min-resource`). For the `min-resource` configuration, SyNAPSE almost halves the SRAM utilization, reduces Map RAM utilization by 5× and barely uses the other switch resources when compared to the other configurations. This is particularly useful when multiple applications share the same switch. Operators may choose to optimize a performance-critical application for throughput and a less performance-critical one for resource utilization.

Summary: There is no “one-size fits all” solution when designing NF accelerators. Different goals ask for different offloading approaches, raising the development challenge. These preliminary experiments already showcase the power of SyNAPSE in delivering different performance objectives from the *same* high-level implementation. We are currently working on expanding the platforms that SyNAPSE supports, implementing other more complex NFs, and developing automated (rather than hand guided) search.

5 Discussion

We have demonstrated component-based synthesis as a promising approach to deploying ‘write-once, run anywhere’ dataplane programs for heterogeneous architectures. Our prototype, SyNAPSE, uses interchangeable, abstract components with performance- and resource-tuned implementations. Before concluding, we discuss SyNAPSE in context with related approaches and discuss future work.

Network Synthesis: Recent work has been using synthesis in the scope of traditional SDN and network configuration [15, 17, 33]. This is in contrast to and complementary of SyNAPSE, which synthesizes network functions.

Related Synthesizers: We argue that SyNAPSE’s form of component-based synthesis [13] fits in a sweet spot of complexity and abstraction. Traditional sketch-based program synthesis approaches (e.g., [26]) explore fine-grained changes in code, typically using a template to restrict the search space. This poses a challenge, as the system either has to limit the

Table 1: Resources used by different implementations.

Resource Type	Min. resources	Min. CPU load	Max. throughput
Hash bits	2.2%	11.1%	11.1%
Hash Dist Unit	0%	34.7%	34.7%
SRAM	16.7%	29%	29%
Map RAM	5.6%	30.2%	30.2%
TCAM	0.0%	0.0%	0.0%
VLIW	1.3%	9.6%	9.6%
Meter ALU	0%	25%	25%
Match Crossbar	3.3%	10.1%	10.1%

flexibility of potential solutions, or deal with an intractably large search space that represents nearly all possible programs. By using components as the key abstraction for space exploration, we constrain the search space (avoiding state space explosions), while imposing only practical restrictions on the space of possible designs. Other NF frameworks have used rule-based translation [19, 31], instead of synthesis. The benefit of SyNAPSE relative to these approaches is that rule-based translation only proposes one solution, while SyNAPSE can explore different deployment options and consider trade-offs.

Partitioning: Sonata [11] also tackles the problem of partitioning programs between CPUs and PISA switches, but for network telemetry queries. SyNAPSE handles more general NFs, requiring a more powerful approach like search.

Future Work: Any good heuristic will perform some form of performance prediction, a task which is a research area in its own right [12]. Our search space exploration must not only account for predicted performance of different tasks on different hardware platforms, but must also predict the *transition cost* of transferring packets from one platform to another – another aspect of performance we will need to model. Further, if our predictions have errors, we will have to be careful to understand how errors in our heuristics lead to sub-optimal execution plans (*i.e.*, is it the case that mild errors in heuristics can lead to very bad execution plans? Or that very bad errors in heuristics can still lead to fairly good execution plans?) This task of designing heuristics will become increasingly complex as we add more hardware platforms to SyNAPSE beyond the PISA and x86 platforms which we already support, such as FPGAs, Network Processing Units (NPUs), or Infrastructure/Data Processing Units (IPUs/DPUs).

Acknowledgments

We are grateful to our shepherd, Jediah McClurg, and the anonymous SOSR reviewers. This work was supported by the SyNAPSE CMU-Portugal/FCT project (CMU/TIC/0083/2019), the uPVN FCT project (PTDC/CCI-INF/30340/2017), INESC-ID (via UIDB/50021/2020), and the Intel/VMware 3D FPGA Academic Research Center. F. Pereira is supported by the FCT scholarship PRT/BD/152195/2021.

References

- [1] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 267–280.
- [2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* (jul 2014).
- [3] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*.
- [4] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
- [5] Jian Chen, Xiaoyu Zhang, Tao Wang, Ying Zhang, Tao Chen, Jiajun Chen, Mingxu Xie, and Qiang Liu. 2022. Fidas: Fortifying the Cloud via Comprehensive FPGA-Based Offloading for Intrusion Detection: Industrial Product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*.
- [6] Xiang Chen, Hongyan Liu, Dong Zhang, Zili Meng, Qun Huang, Haifeng Zhou, Chunming Wu, Xuan Liu, and Qiang Yang. 2022. Automatic performance-optimal offloading of network functions on programmable switches. *IEEE Transactions on Cloud Computing* (2022).
- [7] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargatik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. DRMT: Disaggregated Programmable Switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*.
- [8] Intel Corporation. 2022. Intel Tofino 3. <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html>
- [9] ETSI. 2012. Network Functions Virtualisation - White Paper. http://portal.etsi.org/NFV/NFV_White_Paper.pdf
- [10] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 435–450.
- [11] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*. 357–371.
- [12] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. 2019. Performance Contracts for Software Network Functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 517–530. <https://www.usenix.org/conference/nsdi19/presentation/iyer>
- [13] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (Cape Town, South Africa) (ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 215–224. <https://doi.org/10.1145/1806799.1806833>
- [14] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.
- [15] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. 2013. ZUpdate: Updating Data Center Networks with Zero Loss. *SIGCOMM Comput. Commun. Rev.* 43, 4 (aug 2013), 411–422. <https://doi.org/10.1145/2534169.2486005>
- [16] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*.
- [17] Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. 2015. Efficient Synthesis of Network Updates. *SIGPLAN Not.* 50, 6 (jun 2015), 196–207. <https://doi.org/10.1145/2813885.2737980>
- [18] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 15–28.
- [19] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 85–98.
- [20] Netronome. 2022. Agilio LX SmartNICs. <https://www.netronome.com/products/agilio-lx/>
- [21] R. Pagh and F. F. Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* (feb 2004). Issue 51.
- [22] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*.
- [23] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. 2021. Automated SmartNIC Offloading Insights for Network Functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 772–787.
- [24] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. 2012. Design and Implementation of a Consolidated Middlebox Architecture. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*.
- [25] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*.
- [26] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioglu. 2005. Programming by Sketching for Bit-Streaming Programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (Chicago, IL, USA) (PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 281–294. <https://doi.org/10.1145/1065010.1065045>
- [27] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. 2021. Flightplan: Dataplane disaggregation and placement for p4 programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 571–592.
- [28] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. 2012. Deadline-Aware Datacenter Tcp (D2TCP). In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (Helsinki, Finland) (SIGCOMM '12)*. Association for Computing Machinery, New York, NY,

- USA, 115–126. <https://doi.org/10.1145/2342356.2342388>
- [29] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-Wide Measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*.
- [30] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. 2019. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 275–290.
- [31] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. 2020. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 283–295.
- [32] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.
- [33] Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P. Brighten Godfrey. 2015. Enforcing Customizable Consistency Properties in Software-Defined Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 73–85. <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/zhou>