Determination of Throughput Guarantees for Processor-based SmartNICs

Anonymous Author(s)

ABSTRACT

Programmable network devices are on the rise with many applications ranging from improved network management to accelerating and offloading parts of distributed systems. Processor-based SmartNICs, match-action-based switches, and FPGA devices offer on-path programmability. Whereas processor-based SmartNICs are much easier and more versatile to program, they have the huge disadvantage that the resulting throughput may vary strongly and is not easily predictable even to the programmer. We want to close this gap by presenting a methodology which, given a SmartNIC program, determines the achievable throughput of this SmartNIC program in terms of achievable packet rate and bit rate. Our approach combines incremental longest path search with SMT checks to establish a lower bound for the slowest satisfiable program path. By analyzing only the slowest program paths, our approach estimates throughput bounds within a few seconds. The evaluation with our prototype on real programs shows that the estimated throughput guarantees are correct with an error of at most 1.7% and provide a tight lower bound for processor- and memory-bottlenecked programs with only 8.5% and 18.2% underestimation.

1 INTRODUCTION

Data plane programmability promises the ability to add and change functionality on general-purpose network devices. Data plane programs are used in large-scale deployments to provide functionality such as load-balancing [37], DoS-traffic-scrubbing [1], and offloading packet processing from hypervisors [17]. More examples can be found in scientific literature ranging from in-network caching [30] to offloading parts of distributed systems such as Paxos [11], and accelerating machine learning within the network [32, 53, 54].

General-purpose data plane programmability bears the risk of slow programs causing bad throughput. Therefore, match-action pipelines in programmable switches were created to process packets at a fixed packet rate [3]. Match-action pipelines, however, come at the cost of complicated programming languages and reduced expressiveness [20, 24].

Another option are FPGA-based SmartNICs, as these also allow for data plane programmability with a fixed packet rate. However, FPGA NICs cost at least 8× the price of a regular NIC and require a dedicated team of hardware experts [4, 17] to write programs in hardware description languages. FPGAs can be used to implement a processor which is then much easier to program [4] but no longer processes packets at a fixed rate and is less performant than a hardware processor.

Processors are the common target when programming and allow for rich computation and control flow. For example, the Netronome Agilio CX SmartNIC can be programmed in C using a BPF/XDP toolchain [26, 29]. Although BPF limits the number of executed instructions per packet, the resulting throughput is not obvious [26] and can greatly vary between different packets processed by the same program. Measuring the throughput with a traffic trace can give some idea about the performance of a program, but does not help in predicting the performance in case the traffic changes. We want to close this gap in providing a methodology that determines throughput guarantees for processor-based SmartNICs.

Devices such as switches and NICs have bottlenecks which can be well described in terms of achievable throughput. Whenever the rate of incoming (packet-)data exceeds the throughput bottleneck, congestion forms that induces queuing delay and packet drops that then cause bad network performance. Device-induced latency on a fully loaded SmartNIC is dominated by queuing behavior [25, 33] instead of program execution time. We focus on throughput instead of latency and present a methodology to determine a lower bound for the achievable packet and bit rate of a program.

A program developer or network operator can use our fully automated approach to derive the worst-case guaranteed throughput of a program. If this guaranteed throughput is high enough to not cause any congestion, the program can be safely executed on the data path. In case the throughput of the analyzed program does not yet meet the intended demand, she can try a different program variant or further optimize the worst-case which is identified by our approach.

Throughput guarantees are related to the worst-case execution time which is a well-established field of research (see [58] for an overview) and is a hard problem for general programs on typical processors. Packet processing programs are simpler to analyze, since they typically have no unbounded loops [28, 52, 60]. Existing packet processing performance analysis work [28, 48] targets general purpose processors and determine only rough estimates such as the number of executed instructions and number of cache misses. They do not identify the worst-case [48] or require exhaustive symbolic execution [28] which results in unfeasibly long analysis times. We instead target a SmartNIC without memory caches, analyze throughput instead of execution time, can determine both packet rate and bit rate guarantees, and achieve short analysis time due to incremental path enumeration.

To achieve short analysis time, we only analyze the slowest program paths. However, some paths cannot be triggered by any packet and are therefore irrelevant for the achievable throughput. Our approach is based on enumerating program paths ordered from the slowest path to the fastest path and uses satisfiability checks to exclude the unsatisfiable slowest paths. With incremental enumeration, the analysis can already be stopped on the first satisfiable path without enumerating all paths, resulting in short analysis time. In case this analysis time is still too long, e.g., because of path explosion, an incrementally improving lower bound for the throughput guarantee is produced with each enumerated unsatisfiable path. If one waits until the slowest satisfiable path is identified, our approach additionally yields an example packet and memory assignment which can then be used to measure the worst-case throughput on a real deployment.

We implemented a prototype that analyzes BPF/XDP programs compiled for the Netronome Agilio CX SmartNIC. The evaluation on real programs shows that a first lower throughput bound can be determined within 23.6 s and can be improved by up to 44% within 101.9 s. Throughput measurements show an error of up to 1.7% and a tight lower bound for processor- and memory-bottlenecked programs with only 8.5% and 18.2% underestimation. Our prototype yields useful results for real programs in a timely manner.

Structure. We start by explaining the targeted SmartNIC's architecture in § 2 and subsequently give an overview on our throughput analysis approach in § 3. Then, § 4 describes the per-path throughput capacity heuristics followed by § 5 which presents our incremental ordered path enumeration approach. § 6 evaluates the accuracy and analysis time of our prototype. Finally, we discuss our approach in § 7 followed by related work in § 8 and a conclusion in § 9.

2 PROCESSOR-BASED SMARTNICS

We analyze BPF/XDP [26, 29] programs executed on the Netronome Agilio CX 2x40 GbE SmartNIC. The Netronome Flow Processor (NFP) on this NIC is similar to its predecessor, the Intel IXP network processor. Both have been investigated in previous performance works [6, 10, 21, 23, 25, 46, 49]. Our work is based on the NFP's predictable cycle costs and the program properties ensured by the BPF/XDP toolchain.

Islands. As shown in Figure 1, the NFP is organized into islands which communicate over a high-throughput switching fabric [29, 39, 40]. Some islands contain processing cores whereas others contain special functions such as Ethernet, PCIe, or a transactional memory engine with DRAM.

Many Simple Cores. Packet processing is parallelized onto a huge number of small cores lacking features such as branch prediction, out of order execution, and integer division [42]. Instead of caches, memory access latency is masked by cooperative hyper-threading whereby a thread may yield when waiting for a memory response.

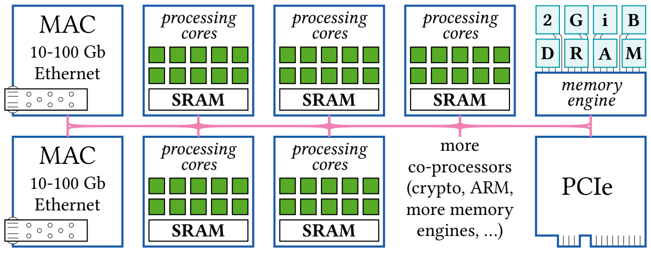


Figure 1: The Netronome Flow Processor architecture.

An Explicit Memory Hierarchy. The NFP has different kinds of memory with varying access latencies [39]. Each processing core has fast access to its own instruction and data memory, medium latency when accessing the SRAM shared by all cores of its island, and some larger latency when waiting for a response from the memory engine which handles DRAM access. Unlike when using a cache hierarchy, pointers always explicitly encode which memory to access.

BPF/XDP on NFP. We analyze NFP Programs produced by the BPF/XDP [26, 29] toolchain since it supports high-level programming languages such as C and P4 and compiles programs to both x86_64 and NFP bytecode [29, 45]. A simplified example program is shown in Figure 2. The Linux kernel loads BPF/XDP programs onto the NIC and verifies program termination by calculating loop bounds and verifies that packet memory accesses are preceded by packet size checks [26]. The NIC's firmware [44] accepts packets over Ethernet and distributes them to 50 processing cores where the BPF/XDP program is invoked for each packet. The program may modify an initial part of the packet in the island's SRAM, may access permanent state in the shared DRAM, and finally decides whether to drop a packet, to transmit it over Ethernet, or forward it over PCIe to the host.

Our goal is, given such a BPF/XDP program compiled to NFP bytecode, to determine a guaranteed throughput that the NIC will always achieve. We, therefore, estimate and compare the amount of processing and DRAM access of a program to identify the program-specific bottleneck throughput.

3 THROUGHPUT ANALYSIS

We want to establish throughput guarantees for SmartNIC programs to enable program developers and network operators to assess whether a given program on a given SmartNIC can achieve the required bit or packet rate. We do this with a fully automated analysis for a program's worst-case throughput capacity. Establishing a lower bound for the throughput

```
1 int main(pkt) {
2     if (pkt.size < 100)
3         return XDP_DROP;
4     if (pkt[ethtype] == ETH_IPv4)
5         atomic_inc(&ip4_counter, 1);
6     if (pkt[ethtype] == ETH_IPv6)
7         for (i = 0; i < 10; i++) nop();
8     return XDP_PASS;
9 }
```

Figure 2: A simplified example of a BPF/XDP program.

capacity boils down to identifying which program path takes the longest time to execute.

Program Paths. The execution time and therefore throughput capacity fundamentally depends on the program path (i.e., the list of instructions and their execution time) that is imposed by the program's structure, the packet's as well as the memory's content. For example, when the program from Figure 2 receives an IPv4 packet of at least 100 byte, the program path through lines 2→4→5→6→8 is triggered and the throughput capacity depends on the execution time of the instructions on this path. However, looking at the example, we clearly see that it actually has four packet classes (`pkt.size` < 100, IPv4, IPv6, other). Each packet class results in a different program path and thus, different executed instructions, likely having a different throughput capacity. As such, any approach that wants to provide a lower bound on a program's throughput capacity must identify the slowest path through the instructions of a program.

Per-Path Throughput Heuristics. To identify the paths with the lowest throughput capacity, a heuristic is needed which estimates the execution time of instructions. The instruction's execution time on the processing cores is, however, not the only variable throughput limitation on the NFP. Instructions that issue memory operations to the shared DRAM may overload the memory engine. When the memory engine is overloaded, the packet throughput becomes a function of the memory engine's rate of executing memory operations. Depending on the ratio between memory and non-memory instructions, the achievable throughput of a program path is limited by either the execution time on the processing cores or the induced load on the memory engine. By using separate heuristics, the throughput capacity of the processing cores and memory engine can be independently estimated for each program path and then compared to identify the actual bottleneck.

With an overall throughput capacity number for each path, we can identify the path with the lowest throughput capacity independent of the individual path's bottleneck. In our example from Figure 2 we can therefore figure out whether the path through line 5 or the path through line 7 has the lower throughput capacity despite one being memory bottlenecked and the other being processing core bottlenecked.

Impossible Paths. When identifying paths through the program, we may encounter impossible paths that cannot be triggered by any packet. Looking at our example, the path with the highest number of executed instructions (2→4→5→6→7→8) cannot be triggered by any packet since the **`if`** conditions in lines 4 and 6 contradict. If such an impossible path is estimated to yield the lowest throughput capacity, its guarantee is not in itself wrong, however, as this execution can never occur in reality, the throughput bound may be far off from the actual (higher) lowest throughput capacity. As such, checking whether paths are possible has the potential of more closely estimating throughput guarantees.

Packet Sizes. There are two different commonly used metrics for throughput capacity: packet rate and bit rate. Many programs process only small headers independent of the actual packet size. For those programs, a bit rate guarantee is equivalent to a packet rate guarantee multiplied by the Ethernet minimum packet size of 60 byte (without CRC). Longer packets increase the actual bit rate, but cannot be considered for a bit rate guarantee as long as the same program paths can be triggered by small packets. This changes, if the program processes longer headers (e.g., tunneling, IPv6 options) or accesses the payload. Whenever a program successfully checks the packet size to access packet data beyond the 60 byte mark, we can infer that the actual packet size is at least the checked size. We can therefore use this knowledge on the packet sizes to establish higher bit rate guarantees.

In the example from Figure 2, all paths containing 2→4 require a minimum packet size of 100 byte. It is not obvious if the short path 2→3 triggered by a small packet, or one of the longer paths triggered by a longer packet, result in a lower achievable bit rate. To identify the path with the lowest bit rate, both the execution time of paths and the minimum packet size required by paths must be considered. Our approach can be used to analyze either a packet rate guarantee or a bit rate guarantee by ignoring or analyzing minimum packet sizes.

Path Explosion. When searching for the path with the lowest bit or packet rate, a naïve approach would simply enumerate *all* path and check each path for contradicting branch conditions and throughput capacity. However, the number of paths may be too large to enumerate them all. In our example, there are only 2^2 paths through the **`if`**s in lines 4 and 6. Yet, a program with n consecutive **`if`**s may produce 2^n paths rendering naïve enumerations quickly infeasible.

Naturally, we strive to enumerate only as few paths as possible while still producing valid throughput guarantees. By enumerating paths ordered from slowest path to fastest

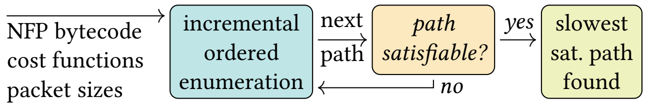


Figure 3: Searching for the slowest satisfiable path.

path (§ 5.1), we get valid throughput guarantees quickly and ignore all paths whose throughput capacity is too high to contribute to the worst-case throughput capacity. As shown in Figure 3, incremental ordered enumeration yields only the slowest paths which are then checked by an SMT solver for contradictions (§ 5.5). In case the enumerated paths are unsatisfiable, incremental ordered enumeration then yields some more paths until a satisfiable path is found. Due to the ordered enumeration, we can stop on the first satisfiable path without enumerating all paths. Thus, the runtime is primarily dominated by proving that low-throughput paths are impossible. This has a further upside, each enumerated path incrementally improves the lower bound, since all paths with a lower estimated throughput capacity have already been shown to be impossible. Our approach, therefore, produces valid intermediate results within a very short analysis time even when a program contains huge numbers of impossible paths with a low throughput capacity.

In the following, we provide details and design rationale for the different steps of our approach. Since the path enumeration builds upon the throughput costs, we start by analyzing the processing (§ 4.1) and DRAM throughput capacity (§ 4.2).

4 PER-PATH THROUGHPUT CAPACITY

Our approach enumerates program paths ordered by the throughput capacity of individual program paths. For that purpose, a heuristic that estimates the execution time of individual instructions can be used to determine the throughput capacity of individual program paths. We start with packet rate throughput since each received packet triggers one program execution. The resulting bit rates are determined at a later step (§ 5.3) by combining these packet rates with program path-specific packet size information.

In an ideal scenario, the SmartNIC manufacturer who has complete knowledge of the inner workings of the SmartNIC would provide a model which perfectly describes the throughput capacities. The documentation [39, 42] of the used SmartNIC contains only incomplete execution timing data and no throughput model. We, therefore, performed measurements on the Netronome Agilio CX SmartNIC to build throughput heuristics of the relevant components.

We identified two components with a throughput capacity which varies based on the executed instructions: the processing cores and the DRAM memory engine. Whenever only one of these components is overloaded, the other component will spend some of its time idling. The actual throughput capacity of a path is the minimum throughput capacity over all components. We therefore analyze a program's throughput capacity separately for each component and then use the minimum. Each instruction is therefore modeled by both a processing core execution time for the case that the processing cores are overloaded and a DRAM execution time for the case that the DRAM memory engine is overloaded. Our approach can be extended to handle more components, but candidates like the per-island SRAM did not show any bottleneck behavior and BPF/XDP programs do not have access to any of the additional NFP co-processors.

Lastly, the NIC itself also has a fixed program-independent throughput capacity limit such as the maximum rate at which the MAC part of the NIC accepts packets or the maximum bit rate of the used Ethernet variant (2x40 GbE in our testbed). For a program to run with the maximum throughput, both the program's processing core throughput capacity and the program's DRAM throughput capacity need to be higher or equal than the fixed program-independent NIC limits.

We start with the processing core throughput heuristic for non-memory instructions followed by DRAM throughput and memory instruction timing.

4.1 Processing Cores Throughput

We want to estimate the throughput capacity of the processing cores for individual program paths. Since programs are executed in parallel on many processing cores, the resulting throughput capacity is influenced by the parallelization onto many cores and the execution time of the program path.

Many-Core Parallelization. The Netronome Agilio CX executes XDP/BPF programs on 50 processing cores. To investigate the impact of parallelization we measure the throughput while varying the number of processing cores. We use programs which do not access any memory, since in this first step we only investigate the processing cores. § 6.1 has more details on how we generate huge numbers of identical packets to always trigger the same program path.

Figure 4 shows the resulting packet rates for two programs, a small program performing a small number of calculations for each packet and a larger program performing more calculations. As can be seen with the black bars showing the 99% confidence intervals, there is only little variation between multiple runs of the same configuration. No configuration exceeds a throughput of 54.4M pkts/s, which was confirmed by Netronome to be roughly the maximum rate at which the MAC part of the NIC can receive packets. Below this limit, the packet rate is strongly proportional to the number of cores, which can be seen by the fitted lines with a resulting R^2 close to 1. Since the throughput is proportional to the number of cores and the clock frequency of the cores is fixed,

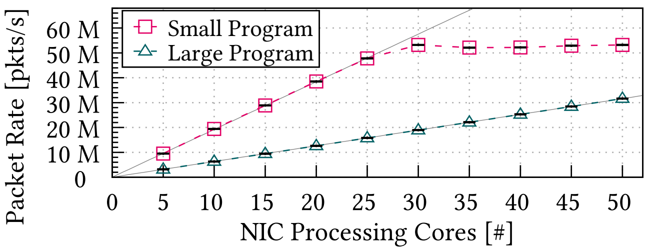


Figure 4: The throughput capacity when using different numbers of NIC cores is limited by the maximum packet rate of the NIC of 54.4M pkts/s. Below that, packet rates are proportional to the number of cores, small program: 1.92M pkts/s/core with $R^2 = 0.999949$; large program: 632k pkts/s/core with $R^2 = 0.999985$.

the throughput capacity can be calculated as:

$$\#cores \times \frac{clock\ frequency}{cycles\ per\ packet}$$

Clock Cycles per Packet. The number of clock cycles that a processing core spends per packet is composed of the instructions executed inside the program and overhead in the firmware when moving from one packet to the next. The NFP reference manual [42] states that most non-memory instructions take a single cycle. The cycle costs of a branch instruction is higher if the branch is taken, but does not depend on previous executions since the NFP has no branch prediction. We confirmed and extended the cycle costs with microbenchmarks to build a cycle-accurate model of the relevant non-memory NFP instructions. Given the instruction trace of a program path, the model gives the number of cycles to execute this program path. To calculate the resulting throughput capacity, we additionally need the number of cycles between returning from the program until the program is invoked with the next packet.

To quantify the per-packet firmware overhead we used the smallest BPF/XDP program, overloaded the NIC with packets, and measured the resulting packet rate. The NIC firmware [44] however contains variable packet processing, as it parses multiple headers to assign packets from the same flow to the same host queue. Since queue selection is also exposed to BPF/XDP programs on NFP, this functionality can be moved to a program (or even be replaced by more advanced queue selection [27]) for which we then can determine a throughput guarantee. We, therefore, removed the queue selection decision from the firmware and thereby obtained a fairly constant per-packet firmware overhead which we found to be independent of packet sizes and content. [1] The per-packet cycle overhead is then calculated by converting the measured packet rate into mean cycles per packet and subtracting the calculated cycle costs of our benchmark program. When combining this overhead with an instruction trace, we can calculate the throughput capacity.

[1]All modifications will be open-sourced upon paper acceptance.

4.2 Memory Access

So far, we have looked at non-memory instructions. To analyze programs that access packet data in the per-island SRAM or permanent state in the shared DRAM, we assess the cycle costs and memory bottleneck of memory instructions.

The closed source variant of the NIC firmware [41] accesses the shared DRAM through a hash table abstraction with hidden code which we cannot analyze, whereas the open source NIC firmware [44] does not support DRAM access from BPF/XDP programs. Since raw memory instructions are easier to analyze, we modified the open source NIC firmware and the NFP Linux kernel driver to expose the shared DRAM as raw memory through BPF array maps. More complex memory access schemes can then be implemented within BPF/XDP programs and will then be analyzed by our approach together with the rest of the program.

Since the NIC's documentation contains only coarse memory latency information [39] and no memory throughput data, we instead derive a throughput capacity heuristic from measurements. As stated before, we observed no bottleneck behavior on the per-island SRAM but a varying shared-DRAM throughput capacity which depends on the executed memory operation and the accessed memory locations.

The observed memory throughput is lowest when spreading the accessed locations by no more then 16 byte and increases by four times when spreading accesses over large ranges. Since we determine throughput guarantees, we must analyze the worst-case which is the case were only a small range of memory is accessed. A factor of up to 4 may cause a huge underestimation of the actually achievable throughput and we are unable to analyze which memory access patterns a program may experience. However, our evaluation (§ 6.1) shows a much smaller gap between estimated worst-case and measured throughput, since the analyzed programs repeatedly access the same memory locations when repeatedly receiving the same packet.

DRAM Throughput Capacity. As shown in Figure 5, we measured the achievable packet rate for small programs which perform different numbers of read operations to the same location in the shared DRAM. When using few processing cores, the processing cores are the bottleneck, as can be seen by the initial proportional increase in packet rate when increasing the number of cores. Once there are enough cores to overload the memory engine with read operations, the packet rate remains constant since the memory engine throughput capacity now dominates the resulting packet rate. The resulting memory throughput, which is calculated

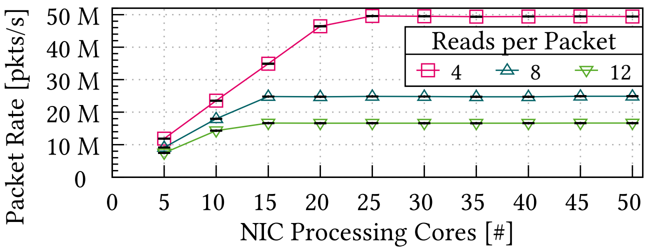


Figure 5: The DRAM bottleneck is observable when enough processing cores are used. The memory engine then performs at a rate of 197.4M ops/s to 199.7M ops/s independent of the number of reads per packet.

by multiplying the packet rate with the reads per packet, is in the range of 197.4M ops/s to 199.7M ops/s for all program variants. We conclude that a read operation incurs a constant worst-case cost on the DRAM memory engine.

We repeated the same measurements with the second DRAM memory operation supported by the BPF/XDP to NFP compiler [45], atomic increment, and observed a constant throughput capacity of 248.5M ops/s to 250.5M ops/s. With the derived DRAM cost functions for memory instructions, we can calculate the DRAM throughput capacity of a program path. The overall throughput capacity of a program path is then limited by the minimum over its DRAM throughput capacity and processing core throughput capacity.

Memory Cycle Costs. Executing memory instructions puts load onto both the memory engine and processing cores. Although the NFP processing cores have no caches, memory instructions still take a variable number of cycles. If the memory engine is not overloaded, an atomic increment takes a single cycle on a processing core since it does not wait for a response from the memory engine. However, reading from DRAM or SRAM pauses execution until the result is available, even when the memory engine is not overloaded. Hyper-threading masks the throughput impact of waiting for a response since another instance of the same program is scheduled. This works well for a single memory access, but can still lead to all threads waiting, if all threads on a core issue memory requests within short succession. We found that the resulting throughput can be estimated well by a minimum number of clock cycles between two memory operations within an instruction trace. We empirically determined the minimum cycles between blocking memory instructions and found different values for per-island SRAM and shared DRAM access.

Our processing core and DRAM throughput capacity heuristics are complete for all instructions issued by the BPF/XDP to NFP compiler and can be used to determine the throughput capacity for all program paths. In the next step, we build upon these heuristics and use the derived cost functions to enumerate paths ordered by achievable bit rate or packet rate and identify or underestimate the slowest satisfiable path.

5 FINDING THE SLOWEST SATISFIABLE PROGRAM PATH

Given that we can estimate the runtime costs of individual program paths, we now need to find the slowest path. However not all paths are actually possible to execute. As such we are looking for the satisfiable path that gives the lowest throughput capacity, which we will simply refer to as the slowest satisfiable path (SSP). This SSP yields a valid throughput guarantee, since all other paths have either a higher throughput capacity or cannot be executed.

5.1 Incremental Sorted Path Enumeration

To mitigate path explosion, we avoid analyzing fast paths, since only the SSP determines the throughput guarantee. Instead, we enumerate paths ordered from lowest to highest throughput and stop analyzing on the first satisfiable path. With each analyzed path, we get an improved lower bound until the SSP yields the final throughput guarantee.

Before identifying the SSP, it is unknown how many impossible paths need to be checked. Enumerating a fixed number of slowest paths may not suffice to find the SSP. Therefore, a mechanism is needed to efficiently enumerate additional paths in case all already enumerated paths are unsatisfiable.

As discussed in the previous section, the SmartNIC has multiple throughput limiting components and we use separate cost functions for each component, e.g., processing core cycle costs and DRAM cycle costs. The resulting throughput capacity is always the minimum over the throughput capacities of the individual components. Because of its simplicity, we choose the incremental longest path algorithm [31] to enumerate paths for a single component ordered by packet rate. We then combine multiple instances of this algorithm for different components and different packet sizes.

The incremental longest path algorithm [31] was initially proposed to find the maximum delay in integrated circuit (IC) designs. Similar to our problem, IC designs have "non-functional" paths which cannot be triggered and therefore do not contribute to the highest possible propagation delay through the IC. The incremental longest path algorithm is suitable for our needs because, after it has already enumerated the n longest paths it can enumerate the $n + 1$ longest path with a time complexity independent of n.

A single instance of this algorithm suffices to determine packet rate guarantees for a single component, e.g., processing cores or DRAM. We continue describing how to transform a program into a graph suitable for this algorithm.

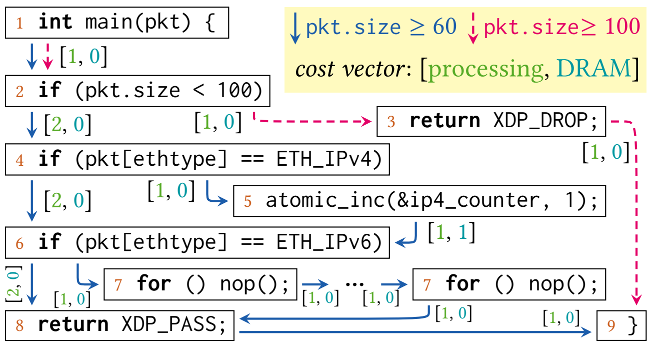


Figure 6: The resulting CFG of the program from Figure 2. A realistic example would use NFP bytecode.

5.2 Preparing a Suitable CFG

To search for the SSP, a control flow graph (CFG) of the analyzed program is needed which is loop-free and has constant costs.

Loop Unrolling. Packet processing programs typically have only bounded loops [28, 52, 60] and the BPF in-kernel verifier only loads NFP programs onto the NIC if it can prove the loop bound [26]. We, therefore, unroll all loops as shown in Figure 6 which has multiple copies of the **for** loop in line 7.

Cost Vector. We search for the SSP according to the processing core cycles and DRAM cycles of individual instructions. Therefore, each edge in Figure 6 has a cost vector although all edges except 5→6 do not incur any DRAM costs. Paths for either the processing cores or DRAM can then separately be enumerated by using a single entry from the cost vectors.

Edge Costs. We use edge costs instead of node costs, since the number of processing cycles to execute a branch instruction depends on the branch result as can be seen at 2→3 and 2→4. This does however not yet suffice to have constant costs since our cost functions for SRAM and DRAM instructions depends on the number of cycles since the previous memory instruction. We overestimate the cycle costs for each edge according to the maximum possible with any path to this edge, which gives a valid underestimated throughput capacity.

The preprocessed CFG can be used to estimate a packet rate guarantee for either processing cores or DRAM. We continue with combining multiple search instances to estimate overall packet rate guarantees.

5.3 Combining Multiple Cost Functions

We separately enumerate paths for processing cores throughput capacity and DRAM throughput capacity and the minimum over both components gives the overall throughput guarantee. Identifying the SSP for each component separately in succession, however, may result in unnecessary analysis work and does not yield valid overall intermediate results. Instead, we simultaneously use multiple instances of incremental longest path search and interleave their results in ascending throughput capacity order.

Multiple Search Instances. We enumerate paths by their overall throughput capacity with the following procedure. For each cost function, a separate search instance is initialized and asked for the slowest path to form an initial set of candidate paths. Although these paths are enumerated using different cost functions, they can be compared by their throughput capacity and the slower path is the path with the lowest overall throughput capacity. In case this path is found to be unsatisfiable, the originating search instance is asked for the next slowest path such that the set of candidate paths again includes a path from each search instance and can yield the next overall slowest path. With this procedure, paths are enumerated according to the overall throughput capacity and each enumerated path gives a valid intermediate result for the overall throughput guarantee.

Each search instance produces each path, but each path should be enumerated only once and only for its bottleneck throughput. In our example from Figure 6, the path 1→2→4→5→6→7...7→8→9 is a slow path for both the processing cores and DRAM, but we are only interested in the bottleneck of this path. Each path is enumerated first for its bottleneck component and later enumerated again for the other components. We, therefore, only consider paths when they are enumerated for their bottleneck and discard them in all other cases. Now, all paths are enumerated once and ordered by their minimum over the processing core and DRAM throughput capacity.

Up to this point, we can determine packet rate guarantees, but not yet bit rate guarantees. Next, we analyze packet size requirements and use even more search instances.

5.4 Enumerating by Bit Rate

Achievable bit rates depend on the cycle costs of a path and on the minimum packet size (MPS) required to trigger the path. Some CFG edges require larger packets (e.g., 2→4), but this packet size information cannot be mapped to constant edge costs. Instead, we determine the set of possible MPSs for a program and then enumerate paths for each distinct MPS by a separate search instance. In our example from Figure 6 one search instance enumerates paths with an MPS of 60 byte and another search instance enumerates paths with an MPS of 100 byte. For each distinct MPS we additionally need separate search instances for processing core and DRAM throughput capacity. Our example needs a total of four search instances to enumerate paths ordered by their overall bit rate capacity.

Packet Size Analysis. For each distinct MPS, only a subset of the CFG edges is needed to cover all paths with that MPS. We statically analyze the MPS requirement for each CFG edge and then collect the set of edges needed for each distinct size. In Figure 6, the edge 2→4 requires a packet of at least 100 byte and additional predecessor and successor edges are needed to cover all paths which have an MPS of 100 byte. In this example, the solid edges are used to enumerate 100 byte paths and the dashed edges to enumerate 60 byte paths. Edge 1→2 is needed for both 60 byte and 100 byte.

Since we use static analysis to determine packet size requirements for edges, we have to underestimate them to produce valid lower bounds for the bit rate guarantee. Therefore, a search instance for a particular MPS may produce some paths which require a larger packet size. Using the smaller size still yields valid lower bounds and the bit rate guarantee is further improved by enumerating additional paths up to a bit rate which matches the actually larger MPS.

Improving Overestimated Costs. We overestimate edge costs and underestimate packet sizes, both of which lead to an underestimation of a path's throughput capacity. We enumerate ordered by this underestimated and the underestimation for a satisfiable path gives a valid throughput guarantee. This underestimation can be further improved by enumerating additional paths. The non-underestimated throughput capacity of a path can be used once all paths with a higher underestimate have been enumerated. Thereby, lower bounds of underestimated paths can be improved by enumerating a few more paths.

With packet size analysis, paths can be enumerated ordered by their achievable bit rate. Each enumerated path is then checked for satisfiability and the first satisfiable path establishes the bit rate guarantee.

5.5 Checking Paths for Satisfiability

Some program paths cannot be triggered by any packet since they contain contradicting branch conditions. We use an SMT solver to check each enumerated path for such contradictions. In case a path is satisfiable, the SMT solver additionally produces an accurate MPS and a minimally sized packet and DRAM assignment to trigger the path.

Symbolic Memory and Pointers. We track register and memory assignments with quantifier-free bitvector and array logic, resulting in branch conditions that depend on a symbolic packet and symbolic DRAM content. The memory region addressed by a symbolic pointer can be ambiguous. Since the BPF in-kernel verifier ensures that pointers always stay within their memory region, we can assume a segmented memory model [2] where no operation on a pointer can change the memory region it points to.

For each satisfiable path, the SMT solver additionally produces a DRAM assignment and minimally sized packet which triggers the path. We evaluate the estimation accuracy by measuring throughput with these example packets.

6 EVALUATION

The evaluation is based on a fully implemented prototype[2] and analyzes real XDP/BPF programs. We use the Z3 [36] SMT solver and enumerate batches of program paths to parallelize satisfiability checking onto CPU cores. We evaluate the estimation accuracy, the time to compute the throughput guarantees, some of the design choices, and use cases.

Analyzed Programs. We estimate throughput and measure throughput on real XDP/BPF programs. The programs shown in Table 1 are a mix of programs well-established in research (the parser from switch.p4 [8] compiled to BPF with p4c-xdp [5]), concepts derived from documentation (Cloudflare DoS [1], QUIC LB [15]) and new programs (RTP a→μ-law [19, 51], DNS Cache [50], Count-Min [9], Path Explosion). Some programs contain bounded loops; the IPv6 variant of the QUIC load-balancer loops over IPv6 options, the RTP a→μ-law transcodes up to 160 bytes of audio payload, and the Count-Min sketch loops over a configurable number of hash functions. Other programs are designed to have their bottleneck at the memory engine; both DNS Cache and Count-Min perform many DRAM lookups and updates. Finally, we created a program to resist our analysis with 2^{64} unsatisfiable paths which are slower than the SSP.

Some of the analyzed programs can exceed the maximum achievable packet rate of the NICs MAC part of 54.4M pkts/s when executed on all 50 processing cores. Although this is usually a desired result when developing a program, it limits our ability to evaluate the estimation accuracy, as we can no longer measure the program's throughput capacity. For this evaluation, we, therefore, estimate and measure throughput capacities of processing core limited programs at 5 processing cores and then scale these numbers to 50 processing cores. We still estimate and measure DRAM throughput limited programs at 50 processing cores at the cost of being unable to measure all satisfiable paths through these programs.

6.1 Estimation Accuracy

To assess the accuracy of our estimates, we measure the throughput of individual program paths.

Testbed. We use a Barefoot Tofino Switch to generate huge numbers of identical packets, similar as proposed by P4pktgen [47]. Each program path is measured separately by repeating a single packet which always triggers this path. For most program paths, the throughput capacity of a single path can be measured by sending more packets than the NIC can handle. We then determine the rate of actually processed packets by reading NIC counters at fixed intervals over 30 s runs and calculating 99% confidence intervals. Due to a bug in the MAC part of the NIC firmware (confirmed by the vendor) we had to measure some of the program paths

[2]The prototype will be open-sourced upon paper acceptance.

Table 1: The throughput guarantees are improved by up to 44% by identifying the SSP and increase by up to 13% by measuring identified paths. The estimated slowest paths are correct with an error of at most 1.0%.

Analyzed Program	Naïve Bound [Bit/s]	Estimated Slowest Sat. Path [Bit/s]	Slowest Measured Path [Bit/s]
switch.p4 (parser)	17.1G	**+44%** 24.7G	**+4.1%** 25.8G±0.05G
Cloudflare DoS	32.1G	**+10%** 35.2G	**-1.0%** 34.7G±0.07G
QUIC LB (IPv4)	22.8G	**+0%** 22.8G	**+2.9%** 23.5G±0.04G
QUIC LB (IPv6)	21.4G	**+29%** 27.6G	**+2.9%** 28.5G±0.05G
RTP a→µ-law	2.97G	**+0%** 2.97G	✓ 2.97G±0.01G
DNS Cache	6.4G	**+41%** 9.0G	**+13.1%** 10.4G±0.02G
Count-Min (5)	21.5G	**+0%** 21.6G	**+2.4%** 22.2G±0.01G
Count-Min (20)	6.0G	**+0%** 6.0G	✓ 6.0G±0.04G
Path Explosion	1.2G	–	–

Table 2: The time it takes to calculate naïve and worst-path throughput guarantees compared to the time it takes to enumerate and check all possible paths.

Analyzed Program	Naïve Bound	Slowest Sat. Path	All Sat. Paths
switch.p4 (parser)	**1.6 s** ±16 ms	**28 s** ±24 ms	**68 s** ±69 ms
Cloudflare DoS	**0.6 s** ±15 ms	**0.9 s** ±14 ms	**2.5 s** ±13 ms
QUIC LB (IPv4)	**0.3 s** ±63 ms	**0.4 s** ±63 ms	**0.5 s** ±65 ms
QUIC LB (IPv6)	**1.1 s** ±21 ms	**33 s** ±55 ms	≥ **1 h**
RTP a→µ-law	**23.5 s** ±57 ms	**102 s** ±57 ms	≥ **4 m**
DNS Cache	**2.5 s** ±25 ms	**38 s** ±64 ms	**13 m** ±0.4 s
Count-Min (5)	**0.2 s** ±1 ms	**0.2 s** ±1 ms	**0.5 s** ±2 ms
Count-Min (20)	**0.8 s** ±4 ms	**1.0 s** ±4 ms	**33 m** ±5.6 s
Path Explosion	**0.5 s** ±4 ms	≥ **47 m**	≥ **47 m**

differently. For these program paths, we shape the rate of transmitted packets to determine the maximum rate the NIC can handle without breaking down.

Per-Path Accuracy. To asses the limits on our estimation accuracy, we measure the throughput of many paths. We, therefore, enumerate not only the SSP but continue enumerating slower paths for one hour, thereby discovering a total of 21 470 measurable paths. The estimate matches the measurement for 9.9% of paths, underestimates 89.6% of paths and is too high for 0.4% of paths. No processing- and memory-bottlenecked paths is underestimated by more than 8.5% and 18.2%. For the paths with a too high estimate, no estimate exceeds the measured throughput by more than 1.7%, possibly caused by inaccuracies in our per-path throughput heuristic. Despite our per-path throughput heuristic being based on measurements, it still produces mostly accurate and tight lower throughput bounds.

Slowest Satisfiable Path. We establish throughput guarantees for programs by identifying the SSP. The estimated SSP is indeed also the slowest measured path for all except one program. For the DNS Cache, the slowest measured path was wrongly estimated to be the seventh slowest path and has a measured bit rate 2.4% lower than the measured bit rate of the estimated SSP. Such inaccuracies are expected, since our example packets do not produce a worst-case memory access pattern. As shown in Table 1, the slowest measured bit rate for DNS Cache is still 13.1% higher than the estimated worst-case throughput capacity. For all analyzable example programs, the estimated worst-case throughput capacity is close to the slowest measured path.

Naïve Lower Bound. For each program, we calculate different throughput guarantees: a naïve lower bound which is the throughput estimate for the slowest, possibly unsatisfiable, path, and a throughput estimate of the SSP. As can be seen in Table 1, this search for the SSP improves the throughput guarantees by up to 44%. However for some programs, the naïve bound cannot be improved, since for these programs the overall slowest path is already satisfiable. Satisfiability checking of paths has the potential of significantly improving throughput guarantees, but is not needed for all programs and prolongs the analysis time.

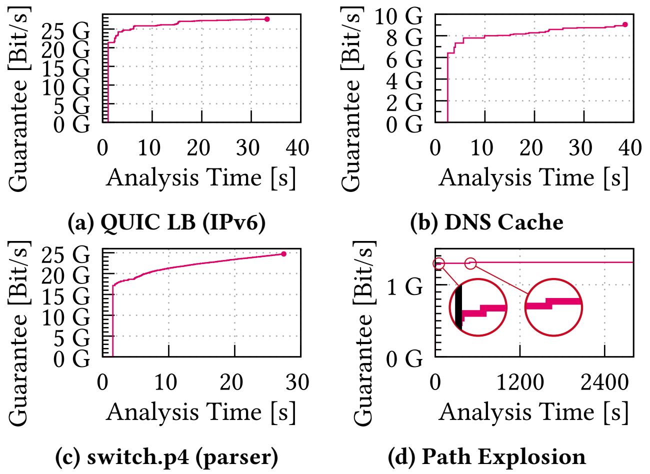
6.2 Analysis Time

For a useful approach, analysis results have to be computed within a reasonable time, even when path explosion happens.

Analysis Setup. We executed our prototype on a desktop computer with an Intel Core i7-7700 CPU with 4 cores (8 threads) and 16 GiB of RAM. Every program analysis was repeated over 20 runs with non-terminating runs being aborted after 1 hour. The results of our analysis time evaluation are realistic since we fully implemented the approach as a working prototype, ran this prototype on real programs, and used a desktop computer with typical performance characteristics.

Analysis Time. As can be seen in Table 2, the naïve bound is computed on all example programs within 23.5 s and except for the Path Explosion program, the SSP is found within 102 s. Analyzing a SmartNIC program takes only little time, enabling developers to regularly check throughput guarantees. The analysis times are so short, it is even feasible to integrate our prototype into regularly executed regression tests.

A major advantage of our approach is the ordered enumeration instead of enumerating all paths. For many programs, we were unable to enumerate all satisfiable paths within an hour or because we ran out of memory before that. Even



(a) QUIC LB (IPv6) (b) DNS Cache (c) switch.p4 (parser) (d) Path Explosion

Figure 7: The throughput guarantee improves until a satisfiable path is found or the the analysis is aborted.

when it is feasible to enumerate all paths, the time to enumerate only the SSP is significantly lower. Note that enumerating additional paths or directly estimating a programmer-defined path is possible and may give further insights.

Path Explosion. On the Path Explosion program, our prototype checked 241 174 paths before running out of memory without having discovered a single satisfiable path. However, the naïve bound, which is a valid throughput guarantee, can always be computed independently of path explosion.

Intermediate Results. In case it takes too long to identify the SSP, ordered enumeration produces valid intermediate results for the throughput guarantee. Each plot in Figure 7 shows one analysis run where a first throughput guarantee is established through the naïve bound and then improved until the SSP is found or the analysis is aborted. If for example, the QUIC LB (IPv6) program needs to process 25 GBit/s, the analysis can already be stopped after 5.7 s instead of 33.2 s. There is however no guarantee that a useful intermediate result is produced in a significantly shorter time, as can be seen with the Path Explosion program.

To summarize, our prototype finds the SSP within minutes on all useful example programs and yields intermediate results before that. In case the SSP cannot be found, the naïve lower bound and additional intermediate results still produce valid throughput guarantees for any program.

6.3 Influence of Design Choices

The analysis time is influenced by several design choices.

Satisfiability Checking. Unlike our approach, symbolic execution checks the satisfiability each time the searcher crosses a branch instruction. We only check the satisfiability of *completed* paths ordered by throughput capacity to avoid costly checking of fast paths. For comparison, we modified our prototype to perform checks on each branch. As shown in Table 3, the time to find the SSP increases on our example programs by at least a factor of ×1.19 and for some programs increases the time from previously minutes to beyond 1 hour, confirming our choice of reducing the checks.

Table 3: Minimum and maximum change over the analyzable examples. Higher numbers are slower.

Alternative Implementation	SSP Analysis Time
Check Satisfiability on each Branch	×1.19 – ≥ 1 h
No Static Analysis	×0.35 – ≥ 1 h
Separate Processor & DRAM Analysis	×1.07 – ×2.36
Packet Rate Analysis	×0.02 – ×1.37

Static Analysis. We use static analysis to remove impossible CFG edges and to determine the per-edge minimum packet size. Some example programs do not benefit from this static analysis and can be analyzed faster without it, for other programs it becomes unfeasible to analyze them in a reasonable time without static analysis. Static analysis, therefore, is an important step in our approach.

Combined Processor & DRAM Anlysis. Instead of interleaving the throughput capacity analysis for the processing cores and DRAM, these could be analyzed separately in succession. Separate analysis not only has the disadvantage of no valid intermediate results but also is slower in all cases.

Packet- vs. Bit-Rate Analysis. Depending on the program and use case, one might be interested in packet rates or bit rates. Packet rate analysis is simpler since it is independent of packet size information. Packet rate analysis can indeed be significantly faster, but this is not the case for all programs.

6.4 Use Cases

Our approach can have additional uses while developing a SmartNIC program.

Program Optimization. We were unsatisfied with the bit rate of the RTP a→μ-law transcoder. Upon inspecting where the SSP spends most of its execution time, we were able to create a program variant with identical behavior but a 58% higher bit rate. Our approach, therefore, can be used as a tool to aid the optimization of programs, although this still requires human effort.

Program Parametrization. The Count-Min program overcounts the number of packets for individual network flows. When increasing the number of different hash functions used for the count-min sketch, the counting accuracy increases at the cost of a decreased achievable packet rate. We, therefore, analyze differently parameterized variants of this program. As shown in Figure 8, the Count-Min program achieves a

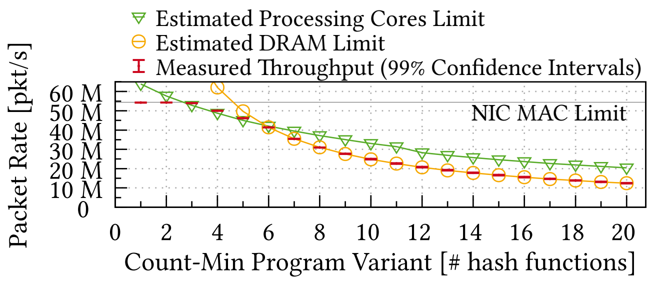


Figure 8: Packet rate estimations and measurements for different variants of the Count-Min flow counter.

perfect throughput with up to 2 hash functions, is processing core limited up to 5 hash functions, and DRAM throughput limited beyond. A developer of such a program can use our approach to analyze the accuracy vs. throughput tradeoff and better decide on a suitable parametrization.

7 DICUSSION & FUTURE WORK

Our approach works quite well, but can still be improved.
Analysis Time. The time to find the SSP is dominated by the work of the satisfiability checker. Improvements in the search strategy should, therefore, focus on reducing the number of satisfiability checks. This could be achieved by reusing satisfiability results for similar paths or reducing the number of to-be-checked paths. Especially, inaccurate minimum packet size estimates may result in incorrect ordering of paths and therefore too many satisfiability checks. Replacing the static analysis of packet size requirements may therefore drastically improve the analysis time for some programs.
Higher Throughput Guarantees. Although our throughput capacity estimates are quite close to our measurements, the estimates will sometimes be much lower than the real throughput. We assume a worst-case DRAM access pattern (§ 4.2), but we are unable to analyze if a path can experience such a bad access pattern and our generated example packets sometimes do not match this bad access pattern. The throughput guarantees could, therefore, be further improved by analyzing programs for their accessed DRAM locations.

We analyze for minimum packet sizes, but real packets will often be larger. Since our packet size analysis is based on accessed packet memory, typical packet payloads that are forwarded but not accessed, are ignored. Better incorporating actual or typical packet sizes might lead to throughput guarantees which are closer to the actual throughput.
Generalization. This paper focuses on programs using the BPF/XDP toolchain executed on a Netronome SmartNIC. For example, a VPN endpoint cannot be reasonably implemented with XDP as the SmartNICs crypto co-processor is currently not accessible from BPF. Extending our approach to analyze programs written in Micro-C [39] or with the Netronome P4 SDK [43] is in principle possible but requires further work. To identify the SSP, programs need to be split into a part without bounded loops and a main loop which iterates over the received packets. Executing different code on multiple processing cores may further complicate the analysis.
Other SmartNICs. When analyzing programs for other processor-based SmartNICs such as Mellanox Bluefield[35] or Marvel LiquidIO[34], our approach needs to be adapted to their throughput characteristics. As these SmartNICs have memory caches instead of cooperative hyper-threading, a modified throughput heuristic is needed. Ideally, the manufacturer of each SmartNIC would provide a throughput model which is then used by our approach to enumerate paths ordered by throughput capacity.
Network Analysis. SmartNIC programs are not running in isolation but are part of a network of programmable and non-programmable devices and applications and often execute only parts of an application. When automatically splitting programs [57] or NF chains [60] our approach can reason about the performance of program parts. The actual worst-case throughput capacity depends on the behavior and interaction of all devices in the network and can be higher than the minimum over the individual devices. A next step could be the performance analysis of a network of SmartNICs.

8 RELATED WORK

Packet Processing Performance. Packet rate and bit rate estimates have been a concern ever since packets were processed on processors [12] in the beginning of the Internet. An important step towards predictable throughput on general-purpose processors is the packet processing system by Dobrescu et al. [14] for which they can extrapolate the throughput when the number of flows changes. Today, the conventional wisdom to achieve predictable throughput is dominated by fixed or programmable match-action pipelines. We show the possibility of predictable throughput on processors by proposing a methodology to analyze SmartNIC programs for their throughput capacity.
SmartNIC Performance. The packet processing performance of SmartNICs is an established topic and we use the same SmartNIC as several previous publications. Hohlfeld et al. [25] and Hasanin et al. [23] showed that throughput and latency of BPF/XDP and P4 SmartNIC programs can vary greatly and depends on multiple influencing factors. George et al. [21], Dai et al. [10], Wu et al. [59] and Chen et al. [6] optimize SmartNIC programs, but give no guarantee on the resulting performance. Qiu et al. [49] applies a performance model to unported programs to estimate the performance of a potential ported program. Since all these works use traffic traces to estimate the performance, they cannot estimate the throughput for unknown traffic.

We instead determine throughput guarantees by analyzing programs for their worst-case traffic.

Mitigating Performance Problems. Without a throughput guarantee, it is unknown how much overprovisioning is needed to eliminate throughput bottlenecks. There is a line of work which assumes that performance problems will happen and then mitigates them. iPipe [33] dynamically adapts the offloaded portion of a program, but can only react once it observes an overload. FairNIC [22] partitions the SmartNIC resources among multiple programs, giving each program exclusive access to a subset of processing cores and caches, thereby limiting an overload to a single program. Unlike these approaches, we can tell beforehand whether an overload may happen and how much SmartNIC resources are needed to eliminate throughput bottlenecks.

Non-Performance Program Analysis. Formal methods such as symbolic execution have successfully been applied to packet processing programs to analyze non-performance properties such as finding bugs [13, 47, 52, 55], verifying reachability [13, 56] and proving correctness [16, 18, 38, 61]. These approaches rely on similar program properties as our approach, e.g., no unbounded loops, and their success shows that it is easier to analyze packet processing programs in comparison to many other programs.

Performance Analysis. We analyze SmartNIC programs for their worst throughput capacity. This is similar to worst-case execution time analysis which is a well-established research field [58] and is hard for general programs on general-purpose processors. Our problem is easier because we analyze throughput instead of latency, because packet processing programs are sufficiently restricted, and because the targeted SmartNICs have comparably simple processing cores.

Our approach has similarities with using symbolic execution for execution time analysis of packet processing on general-purpose processors. Pedrosa et al. [48] identify slow program paths but cannot identify the slowest path, despite significantly longer analysis times compared to our approach. Chipounov et al. [7] exemplarily analyze the longest path through the Apache HTTP Server's URL parser and Rishabh et al. [28] analyze the relationship between the worst-case execution time and parameters such as table occupancy in packet processing programs. Both of them enumerate *all paths* through a program, which is infeasible for many programs. Rishabh et al. [28] suggest restricting the search space by adding additional constraints on the input packet. Performance results for a constrained input can however not be generalized for packets beyond these constraints. In contrast to this, our approach often runs faster with an unconstrained input, since we stop on the first satisfiable path. All these approaches use coarse metrics such as the number of executed instructions and cache misses which cannot easily be mapped to throughput.

9 CONCLUSION

The achievable packet and bit rate of a SmartNIC program is not obvious and varies between different packets and triggered program paths. SmartNICs are easier to program, whereas programmable match-action pipelines and FPGAs can provide a guaranteed packet rate. We want to provide similar guarantees to SmartNICs by analyzing programs for their guaranteed packet rate and guaranteed bit rate. With our approach, a program developer or network operator can determine whether a SmartNIC program will always achieve the needed throughput. In case the program does not yet achieve this throughput, the program can be further optimized or be parallelized onto the right number of SmartNICs.

Different packets trigger different paths through a program. We analyze the guaranteed throughput by identifying or underestimating the slowest program path. We only consider satisfiable paths, since a program may have slow paths which contain contradicting branch conditions and therefore cannot be triggered by any packet. An underestimation for the throughput capacity of the slowest satisfiable path therefore gives a throughput guarantee for the complete program. Programs may have huge numbers of paths, such that it is unfeasible to check all paths through a program for satisfiability and their throughput capacity. Instead, we incrementally enumerate paths from slowest to fastest and stop analyzing on the first satisfiable path. Our prototype determines throughput guarantees for real programs with an error of at most 1.7% and provides tight lower bounds for the processor- and memory-bottlenecked programs with only up to 8.5% and 18.2% underestimation.

We enable developers and network operators to determine whether a program meets the throughput requirements. When integrated into the development toolchains for SmartNIC programs, the developer can get rapid feedback on the throughput capabilities and can iterate on optimizations until the requirements are met. When used for automatic regression tests, changes which lead to undesirable throughput are caught without impacting the production network.

With our throughput guarantees, SmartNICs can be used with the same determinism as programmable match-action pipelines and FPGAs. This enables a step towards more freely programmable switches based on processors. A network operator does not need to fear throughput problems if the used program has an adequate throughput guarantee. Typical programmable match-action pipelines have a fixed packet rate and allow only few processing steps, even on large packets. However, large packets take longer to transmit and therefore allow for more processing time before the next packet arrives. With our approach, a processor-based switch can perform many operations on large payloads and still meet the required bit rate guarantee.

REFERENCES

[1] Gilberto Bertin. 2017. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In *NETDEV 2.1*.

[2] Borzacchiello, Luca and Coppa, Emilio and Cono D'Elia, Daniele and Demetrescu, Camil. 2019. Memory models in symbolic execution: key ideas and new thoughts. *Software Testing, Verification and Reliability* 29, 8 (Dec. 2019), 35 pages. https://doi.org/10.1002/stvr.1722

[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 2013 ACM SIGCOMM Conference (SIGCOMM '13)*. ACM. https://doi.org/10.1145/2486001.2486011

[4] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2020. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association. https://www.usenix.org/conference/osdi20/presentation/brunella

[5] Mihai Budiu and William Tu. 2020. Backend for the P4 compiler targeting XDP. https://github.com/vmware/p4c-xdp.

[6] Michael K. Chen, Xiao Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, and Roy Ju. 2005. Shangri-La: Achieving High Performance from Compiled Network Applications While Enabling Ease of Programming. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM. https://doi.org/10.1145/1065010.1065038

[7] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM. https://doi.org/10.1145/1950365.1950396

[8] P4 Language Consortium. 2020. P4_16 reference compiler. https://github.com/p4lang/p4c.

[9] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75. https://doi.org/10.1016/j.jalgor.2003.12.001

[10] Jinquan Dai, Bo Huang, Long Li, and Luddy Harrison. 2005. Automatically Partitioning Packet Processing Applications for Pipelined Architectures. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM. https://doi.org/10.1145/1065010.1065039

[11] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: Consensus at Network Speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. ACM. https://doi.org/10.1145/2774993.2774999

[12] Donald Watts Davies, K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. 1967. A Digital Communication Network for Computers Giving Rapid Response at Remote Terminals. In *Proceedings of the First ACM Symposium on Operating System Principles (SOSP '67)*. ACM. https://doi.org/10.1145/800001.811669

[13] Mihai Dobrescu and Katerina Argyraki. 2014. Software Dataplane Verification. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/dobrescu

[14] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. 2012. Toward Predictable Performance in Software Packet-Processing Platforms. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12) (NSDI 12)*. USENIX Association. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/dobrescu

[15] M. Duke and N. Banks. 2020. QUIC-LB: Generating Routable QUIC Connection IDs. https://tools.ietf.org/html/draft-ietf-quic-load-balancers-03.

[16] Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2019. Dataplane equivalence and its applications. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association. https://www.usenix.org/conference/nsdi19/presentation/dumitrescu

[17] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*. USENIX Association. https://www.usenix.org/conference/nsdi18/presentation/firestone

[18] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Uncovering Bugs in P4 Programs with Assertion-Based Verification. In *Proceedings of the Symposium on SDN Research (SOSR '18)*. ACM. https://doi.org/10.1145/3185467.3185499

[19] G.711 1988. *Pulse Code Modulation (PCM) of Voice Frequencies*. ITU-T Recommendation.

[20] Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. 2020. Challenging the Stateless Quo of Programmable Switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20)*. ACM. https://doi.org/10.1145/3422604.3425928

[21] Lal George and Matthias Blume. 2003. Taming the IXP Network Processor. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM. https://doi.org/10.1145/781131.781135

[22] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. 2020. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. 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 (SIGCOMM '20)*. ACM. https://doi.org/10.1145/3387514.3405895

[23] Hasanin Harkous, Michael Jarschel, Mu He, Rastien Pries, and Wolfgang Kellerer. 2019. Towards Understanding the Performance of P4 Programmable Hardware. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '19)*. IEEE. https://doi.org/10.1109/ANCS.2019.8901881

[24] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. 2021. A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research. https://arxiv.org/abs/2101.10632.

[25] Oliver Hohlfeld, Johannes Krude, Jens Helge Reelfs, Jan Rüth, and Klaus Wehrle. 2019. Demystifying the Performance of XDP BPF. In *2019 IEEE Conference on Network Softwarization (NetSoft) (NetSoft 2019)*. IEEE. https://doi.org/10.1109/NETSOFT.2019.8806651

[26] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The

EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. ACM. https://doi.org/10.1145/3281411.3281443

[27] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. 2019. Mind the Gap: A Case for Informed Request Scheduling at the NIC. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets '19)*. ACM. https://doi.org/10.1145/3365609.3365856

[28] 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. https://www.usenix.org/conference/nsdi19/presentation/iyer

[29] Nic Viljoen Jakub Kicinski. 2016. eBPF Hardware Offload to SmartNICs: cls bpf and XDP. In *Netdev 1.2*.

[30] Xin Jin, Xiaozhou li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM. https://doi.org/10.1145/3132747.3132764

[31] Sandip Kundu. 1994. An incremental algorithm for identification of longest (shortest) paths. *INTEGRATION* 17, 1 (1994), 25–31. https://doi.org/10.1016/0167-9260(94)90018-3

[32] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21) (NSDI '21)*. USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/lao

[33] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading Distributed Applications onto SmartNICs Using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. ACM. https://doi.org/10.1145/3341302.3342079

[34] Marvell. 2020. Marvell® LiquidIO™ III. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf.

[35] Mellanox. 2019. BlueField SmartNIC for Ethernet. https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[36] Microsoft Research. 2019. The Z3 Theorem Prover. https://github.com/Z3Prover/z3/tree/z3-4.8.7.

[37] Usama Naseer, Luca Niccolini, Udip Pant, Alan Frindell, Ranjeeth Dasineni, and Theophilus A. Benson. 2020. Zero Downtime Release: Disruption-Free Load Balancing of a Multi-Billion User Website. 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 (SIGCOMM '20)*. ACM. https://doi.org/10.1145/3387514.3405885

[38] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association. https://www.usenix.org/conference/osdi20/presentation/nelson

[39] Netronome. 2014. The Joy of Micro-C. https://open-nfp.org/documents/48/the-joy-of-micro-c_fcjSfra.pdf.

[40] Netronome. 2018. Benefits of a Composable Silicon Architecture. https://www.netronome.com/media/documents/WP_Composable-Architecture.pdf.

[41] Netronome. 2019. Agilio eBPF 2.0.6 - extended Berkeley Packet Filter. https://help.netronome.com/support/solutions/articles/36000050009-agilio-ebpf-2-0-6-extended-berkeley-packet-filter.

[42] Netronome. 2019. Netronome ® Network Flow Processor 6xxx NFP SDK version 6 Flow Processor Core Programmer's Reference Manual.

[43] Netronome. 2019. P4 Data Plane Programming for Server-Based Networking Applications. https://www.netronome.com/media/documents/WP_P4_Data_Plane_Programming.pdf.

[44] Netronome. 2020. CoreNIC: a flexible SR-IOV SmartNIC firmware implementation supporting BPF and stateless offloads. https://github.com/Netronome/nic-firmware.

[45] Netronome. 2020. Netronome Flow Processor (NFP) Kernel Drivers. https://github.com/Netronome/nfp-drv-kmods.

[46] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM. https://doi.org/10.1145/3230543.3230560

[47] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. 2018. P4pktgen: Automated Test Case Generation for P4 Programs. In *Proceedings of the Symposium on SDN Research (SOSR '18)*. ACM. https://doi.org/10.1145/3185467.3185497

[48] Pedrosa, Luis and Iyer, Rishabh and Zaostrovnykh, Arseniy and Fietz, Jonas and Argyraki, Katerina. 2018. Automated Synthesis of Adversarial Workloads for Network Functions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM. https://doi.org/10.1145/3230543.3230573

[49] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. 2020. Clara: Performance Clarity for SmartNIC Offloading. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20)*. ACM. https://doi.org/10.1145/3422604,.3425929

[50] RFC1035 1987. *DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION*. Internet Requests for Comments.

[51] RFC7655 2015. *RTP Payload Format for G.711.0*. Internet Requests for Comments.

[52] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. 2020. Gauntlet: Finding Bugs in Compilers for Programmable Packet Processing. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association. https://www.usenix.org/conference/osdi20/presentation/ruffy

[53] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets 2017)*. ACM. https://doi.org/10.1145/3152434.3152461

[54] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21) (NSDI '21)*. USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/sapio

[55] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 Programs with Vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM. https://doi.org/10.1145/3230543.3230548

[56] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2016. SymNet: Scalable Symbolic Execution for Modern Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM. https://doi.org/10.1145/2934872.2934881

[57] 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) (NSDI '21)*. USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/sultana

[58] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems* 7, 3, Article 36 (May 2008), 53 pages. https://doi.org/10.1145/1347375.1347389

[59] Qiang Wu and Tilman Wolf. 2009. Runtime resource allocation in multi-core packet processing systems. In *2009 International Conference on High Performance Switching and Routing (HPSR '09)*. IEEE. https://doi.org/10.1109/HPSR.2009.5307422

[60] Jane Yen, Jianfeng Wang, Sucha Supittayapornpong, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. 2020. Meeting SLOs in Cross-Platform NFV. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '20)*. ACM. https://doi.org/10.1145/3386367.3431292

[61] 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 (SOSP '19)*. ACM. https://doi.org/10.1145/3341301.3359647

A ARTIFACTS

Artifacts are available at https://zenodo.org/record/TODO or https://github.com/johannes-krude/nfp-pred-artifacts. These can be used to **repeat** the full evaluation and can be **reused** to analyze new XDP/BPF programs. Included is the source code and documentation of the main approach, our modifications to the SmartNIC device driver and firmware, as well as the infrastructure and raw measurement data from our evaluation accompanied by documentation.

Requirements. It suffices to use Docker on Linux or a single computer with Ubuntu 20.04 to repeat the provided small evaluation example and analzye the precompiled XDP/BPF programs. To repeat the full evaluation, one needs: three computers, a Netronome Agilio CX 2x40 GbE SmartNIC, an additional 2x10 GbE NIC, and a Barefoot Tofino based EdgeCore Wedge BF100-32X switch. The proprietary compilers to build the NIC firmware and Tofino P4 program are not included and can be obtained from Netronome and Intel.

Implementation. The main approach from the paper is implemented as a tool that determines throughput guarantees by incrementally enumerating programs paths of XDP/BPF programs compiled to Netronome Flow Processor assembly. This tool is implemented in 9600 lines of C++, heavily relies on the SMT solver Z3, and is in part inspired by the KLEE symbolic execution engine. The evaluation infrastructure consists of 4700 lines of Ruby source code with low-level tools written in C and some small additions of Bash, Python, and P4. Our modifications to the SmartNIC device driver and firmware consist of Linux kernel level C and NFP assembly.

Measurements. The evaluation mainly consists of two different types of measurements. During the estimation phase, the approach as described in the paper is executed on example programs to estimate throughput guarantees. All discovered satisfiable program paths are recorded together with an example packet to trigger the path and the time it takes to discover that path. These results from these measurements are presented in Table 1 (columns 2 & 3), Table 2, Figure 7, and Figure 8 (estimates). For Table 3, the throughput estimation is executed again on all example programs but uses different implementation variants.

The second kind of measurements in the evaluation, are measurements of the actual throughput when executing programs on the SmartNIC. These measurements use the example packets from the estimation phase to measure the throughput capacity of each discovered program path. These throughput measurements and their comparison to the throughput estimates are presented in Table 1 (column 4) and Figure 8 (measured throughput). Some additional throughput measurements are shown in Figure 4 and Figure 5.

Repeating the Evaluation from the Paper. The full evaluation takes approximately 10 days and requires a Netronome Agilio CX 2x40 GbE SmartNIC and a Barefoot Tofino-based EdgeCore Wedge BF100-32X programmable switch. To enable partial repetition, we structured the evaluation into smaller steps, some of which take significantly less time and do not require special hardware. All raw measurements gathered during our evaluation are included, to enable repeating each evaluation step independently of the other steps.

When having only a few minutes to spare, one can try the small evaluation example which estimates the throughput bound of the QUIC LB (IPv4) example program and compares these estimates with existing measurements. With some more available time, one can reanalyze all included measurement data and optionally repeat all throughput estimates. In case of having access to the SmartNIC and a Tofino switch, all throughput measurements can be repeated.

Reusing the Implementation for New Programs. Our implementation of the main approach, as well as the measurement infrastructure can be applied to new real XDP/BPF programs. Any XDP/BPF program which adheres to the constraints for NFP offloading and our modified NIC firmware and driver can be analyzed for its worst-case throughput. The implementation supports analyzing for bit rate or packet rate and can be configured to analyze for processing cores throughput, DRAM memory engine throughput, or both. For each enumerated path, an example packet is generated which can be used to compare the estimated throughput capacity to measured throughput. A detailed description is included which explains how to generate the data as presented in Table 1 and Table 2 for new XDP/BPf programs.

All further documentation is included in the `README.md`.