Optimizing Asynchronous Extern Execution in Programmable Software Data Planes

Péter Hudoba
Eötvös Loránd University
Budapest, Hungary
peter.hudoba@inf.elte.hu

Róbert Kitlei
Eötvös Loránd University
Budapest, Hungary
kitlei@elte.hu

Sándor Laki
Eötvös Loránd University
Budapest, Hungary
lakis@inf.elte.hu

Péter Vörös
Eötvös Loránd University
Budapest, Hungary
vopraai@inf.elte.hu

GLOBECOM 2023 - 2023 IEEE Global Communications Conference | 979-8-3503-1090-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/GLOBECOM54140.2023.10436961

***Abstract*—P4 has gained a significant attention as a programming language for describing target-independent packet processing. It supports a diverse hardware and software targets through various architecture models that declare external functions called externs representing target-specific functionalities. These externs may require specific or dedicated resources (e.g., cryptography co-processor, FPGA, etc.) for increased processing speed. In order to process tasks in bulk mode, packet processing may need to be temporarily suspended, while waiting for the function to return. Linear execution of the packet processing pipeline implemented by most P4 software targets cannot efficiently handle such situations. Asynchronous packet processing has been proposed to solve this issue by enabling to serve incoming packets while others are processed by an extern. In this paper, we explore existing approaches for extern execution in software data planes and propose a new lightweight asynchronous method for offloading extern execution to dedicated resources, such as cryptography co-processors, which perform the extern computations. Our analysis show that the propose method can significantly improve the performance of extern execution in various use cases like IPsec, simple encryption and other small tasks and has negligible overhead compared to a prior solution. We also demonstrate that our method has clear benefits on constrained hardware (PcEngines APU single board computer) where the overhead of extern execution has bigger impact on the overall performance and prior approaches are not practical.**

***Index Terms*—P4, software data plane, asynchronous packet processing, IPSec**

I. Introduction

Software Defined Networking (SDN) [1] offers a high level of flexibility and scalability for computer networks by separating the data and control planes and introducing programming abstractions into both layers. While control plane programmability has been extensively studied in the literature, the issue of programmable and portable data planes has become increasingly significant in recent years. To address this, specific programming languages have been developed to enable network developers to describe the entire packet processing in a protocol-independent manner at a high level of abstraction. Among these languages, P4 [2] has gained the most attention in the past couple of years. P4 allows describing packet handling at a high abstraction level, without the need for a deep-level of coding skills and hardware knowledge. This provides a promising solution to the challenges of programmable and portable data planes, allowing for greater efficiency and adaptability in network development.

The language is currently supported by both high-speed hardware (ASICs, SmartNICs) and flexible software data planes. Each target requires a target-specific compiler that generates the binary to be executed in the given environment. In this paper, we focus on the open-source DPDK-based P4 compiler and software switch, called T4P4S [3], that already made some steps towards asynchronous external function (extern) execution. Externs can implement anything that the main P4 language does not support and may require complex computations or dedicated resources (e.g., co-processors, FPGAs). An extern can even perform cryptography operations, image or signal processing or for example a machine-learning method that load balances in a self-learning way. The authors of [4] show a P4 implementation for IPsec where various externs are introduced but their execution is synchronous.

In [5] and [6], the authors show how T4P4S can be used to implement extern functions in an asynchronous way. Their method uses ucontext (Linux's context switching routines, see man ucontext(3)) to save the actual state of the processed packet before sending it to the dedicated resource (e.g., a hardware accelerator) performing the asynchronous task. Their solution, however, is only suitable if the external function has a long run-time. Simple functions such as symmetric encryption are too short for the ucontext-based method as the overhead of asynchronous execution overcomes the advantages of hardware acceleration.

In this paper, we propose a more lightweight approach called longjump for implementing asynchronous extern execution, and show that its overhead is negligible and it outperforms the prior ucontext-based method. Our performance evaluation is based on experimental results considering different case studies including IPSec for VPN tunneling. In addition to a high-speed server setup, we also carried out measurements on a low-cost x86-based single board computer (SBC) where the overhead of asynchronous execution can cause more significant performance degradation. SBCs have grown increasingly popular in recent years, particularly among hobbyists, makers, and small businesses due to their compact form factor and powerful computing capabilities. They are often used in a range of applications, such as media centers, home automation systems, and IoT devices. However, when considering in-network programming, SBCs can also serve as a cost-effective alternative to traditional gateway servers, such

as VPN or secure email gateways. With their ability to support high-speed encryption, SBCs can be used to create VPN gateways that securely encrypt traffic between a company's network and remote devices or networks. Additionally, SBCs can be used to create secure email gateways that encrypt incoming and outgoing email traffic, which is particularly useful in protecting sensitive information and ensuring compliance with data protection regulations. In our experiments, we utilized PcEngines APU4d4, a low-cost DPDK-compatible 4-port SBC, to simulate a real-life small company scenario where the cost of expensive ASICs, such as Intel Tofino, is simply not affordable. With its compact form factor, low power consumption, and powerful computing capabilities, PcEngines APU4d4 has proved to be an effective and cost-efficient solution. Note that the idea of running P4 programs on SBCs was first proposed by the P4Pi (P4 on Raspberry Pi) educational platform [7], but its packet forwarding performance is far below the performance expected by real-world scenarios like a home gateway.

The paper is organized as follows: we show existing approaches for running P4 externs in Section II. Later in Section III, we introduce our new asynchronous execution method called longjump integrated into the T4P4S compiler. In Section IV, we present different case studies focusing on IPsec, a simple encryption functions, and custom external functions to have a wide understanding of the results with functions of various computational complexity. Finally, we give a conclusion in Section V.

II. Known approaches for external functions

There are various ways to execute an external function, which depend on the specific compiler and hardware being used. These external tasks can be any function that may benefit from a speed boost when offloaded to a dedicated co-processor. Cryptography chips are a prime example of this, as they can perform tasks in bulk on an external chip much more quickly. With the discussed APU hardware, an IPsec implementation with the ucontext-based asynchronous approach [5] compared to the synchronous execution used in [4] does not lead to an efficient solution. On this constrained hardware, the asynchronous overhead is more than the advantage of the solution (see Section IV for measurements), thus a more lightweight solution is needed.

In this section, we briefly overview existing approaches for extern execution in software data planes.

A. Non-async mode

The simplest and most commonly used method of executing an external function is an on-demand approach, where the function is executed only when needed, and packet processing continues once it is complete. This approach does not require handling multiple program points or any additional information to manage simultaneous packet processing. Therefore, memory management and code complexity are minimized in comparison to other approaches that have been investigated.

B. Context mode

As demonstrated in [5], another approach to implementing asynchronous packet processing is by using ucontext to temporarily suspend the processing of a packet. In this method, the runner of the P4 program creates a context for the packet that requires asynchronous processing. While the external function is being computed, packet processing is paused and the next packet in the queue is processed. The ucontext saves the program pointer and stack from the appropriate level of the program, allowing the original packet processing to resume seamlessly when the result of the external function is ready. In addition to saving the context, we also need to preserve certain data stored on the heap. This requires additional memory move operations.

III. The new approach

As discussed in the previous section, the ucontext-based asynchronous approach has too large overhead for practical usage on constrained hardware. Therefore, we developed a more lightweight method that provides a good alternative by focusing on the packet descriptor instead of the entire memory stack. The packet descriptor contains crucial information, such as the deparsed data and metadata for a packet. This data is saved into a packet descriptor storage along with a program-state variable. This variable indicates different points in the packet processing program graph, allowing us to save only the essential information, rather than the entire stack. Once the asynchronous task is completed, we continue the packet processing, using the previously calculated external function results. While our approach may require parsing the packet multiple times as a trade-off, it requires less information to be stored overall, making it a more efficient solution for limited hardware. After reparsing the packet, the program-state stored for the packet is used to directly jump to the extern call in the P4 program where the extern's result is available and the P4 pipeline is continued. Note that reparsing the packet may not be needed in all the cases and this step can be avoided. This process is depicted in Figure 1.

Our new approach employs a longjump mechanism instead of the context-saving method. This involves setting two distinct jump points in the program: the first is located in the main loop where packets are initially processed. When a packet arrives that requires external processing, we queue the external task and then jump to the main loop to start processing the next packet. After processing a burst of packets, we check the queue to see if any new results have arrived. If there are new results, we start a new processing of the packet with the updated information. This new process is called the "main async loop". If the P4 program has multiple external function calls, we use the second jump point to continue the main async loop after the second external function call, as the program packet processing is initiated from that point.

Algorithm 1 illustrates the structure of the main asynchronous loop. It sends a batch of external tasks to the co-processor if enough have been collected for one burst. After collecting the results, it reruns the packet processing with the

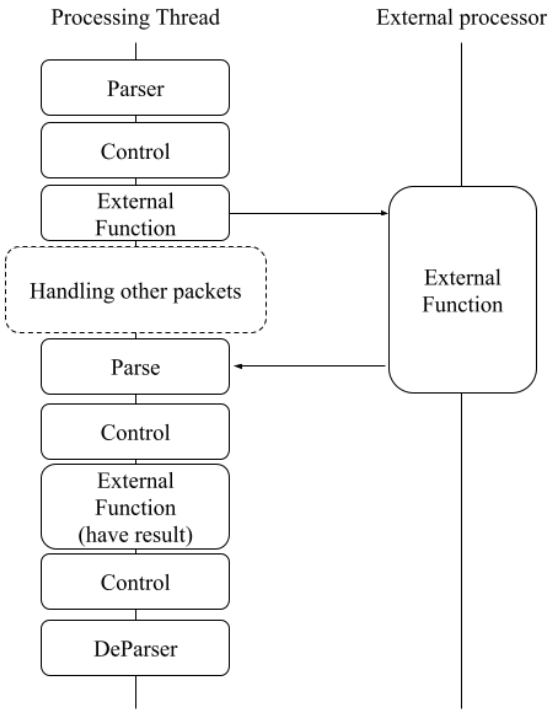


Fig. 1. Sequence diagram of the longjump approach.

newly calculated result. The *burstSize* is a configuration variable that determines how many tasks we collect before sending it to the co-processor. The *pendingExternalTask* variable starts from 0 and keeps track of how many tasks is under processing on the co-processor.

```
taskLength ← length of task queue;
if taskLength ≥ burstSize then
    for each task of queue do
        Enqueue external operation to co-processor;
        Remove from the queue and free task;
    end
    pendingExternalTask ←
      pendingExternalTask + taskLength;
end
if pendingExternalTask > 0 then
    Dequeue external operation results from
      co-processor;
    for each result do
        Get stored packet information corresponding to
          the result;
        Rerun packet processing;
        pendingExternalTask ←
          pendingExternalTask − 1;
    end
end
```

Algorithm 1: Asynchronous task handling main loop

When we enqueue an external operation, we store the packet information in a ring and save the position of the new information in the ring. We append this position to the beginning of the data that will be sent to the external processor. To prevent this position data from being changed, we configure the external processor to start running the function only from the 9th byte. When we dequeue the result, we first remove this position from the beginning of the buffer and restore the stored information from the ring.

The implementation of this new approach can be found in the T4P4S source code, which is available in the project's public repository [8].

IV. Evaluation Case Studies

In this section, we assess the practical performance of the various approaches. We begin by examining the IPsec and simple symmetric encryption cases from multiple perspectives. Then, we analyze how the approaches fare when used for longer external functions to gain a better understanding of overall performance. Finally, we describe our measurements and findings from testing on a more powerful machine setup.

Our experimental setup comprises of two components: a packet generator, which transmits test packets, and a programmable switch, which processes the packets and transmits them to the intended port with the necessary modifications.

For our programmable switch, we choose a popular SBC, the APU4d4, which is manufactured by PC Engines. This board features a quad-core AMD G-Series GX-412TC processor, clocked at 1 GHz, and has 4GB of DDR3 RAM. It also includes four Gigabit Ethernet ports, a USB 2.0 port, and a mini-PCIe slot for expansion. The APU4d4 is well-suited for networking applications, such as routers, firewalls, and access points. The four Gigabit Ethernet ports make it easy to connect to multiple devices, and the board is designed to handle high traffic loads with low power consumption. It is also a very affordable option compared to the much higher-end programmable ASICs or SmartNICs.

Our packet generator for this project was a dedicated server featuring an AMD Ryzen Threadripper 1900X 8-Core Processor and AQC107 NBase-T/IEEE 802.3bz Ethernet Controller. To generate packets, we utilized the DPDK-pktgen software created by Intel, which has the ability to send a high volume of packets while also tracking the number of packets successfully received.

After the switch and pktgen started correctly we collect data for 15 seconds. We trim down the zeroes from the beginning and then drop the first and last 25% of the remaining data to make sure we measure the sustainable performance of the configuration. This 50% data without the leading and trailing zeroes will be used to calculate a mean and that will be considered as throughput in our results This mean will be our main measurement of the performance. Other measure metrics will be filtered similarly to the throughput. Since the variances in all the measurements were negligible, we will not discuss them in this article. It is worth noting that we only present measurement results using packets of size 64 bytes as we got similar results for larger packet sizes.

As a baseline metric, we also measured a simple L2 forwarding P4 program that simply sends the packets to a selected port according to a MAC table. This baseline program resulted in 430k packets/sec forwarding rate in our setup.

A. IPsec

We conducted a performance comparison of our new approach with the approaches discussed in Section II, specifically focusing on the IPsec ESP tunnel mode implementation in T4P4S software. We found that the differences among the operation modes were negligible in terms of effectiveness, so we only focused on one mode of operation of IPsec. Additionally, we did not consider the initialization phase of the IPsec, assuming that the SPI (a unique value that identifies the security association) and sequence number were already known. These pieces of information can be loaded from a table in the P4 code after the initialization phase and were thus not relevant to the optimization.

A recent paper [4] has also discussed implementing IPsec in the P4 language. From the P4 perspective, the synchronous measurements we carried out are similar to their extern execution approach. To encapsulate a packet into an IPsec packet in tunnel mode, we first pad the IP packet to a size that is a multiple of 16 bytes, with the 2-byte sized ESP trailer. We then save the pad length into the second-to-last byte and encrypt it. This is necessary to be able to remove the padding and restore the original data later on. Next, we compose the 8-byte ESP header by setting the SPI and sequence number. These pieces of information can be set in the P4 source code. Finally, we calculate an HMAC for the ESP header and the whole ESP payload and append 12 bytes of the HMAC to the end of the packet. We also refresh the checksum and total length in the new IP header. The structure of the packet transformation can be seen in Fig 2. ESP tunnel mode requires one symmetric encryption and one hash calculation. For symmetric encryption, we use AES, and for HMAC authentication, we use MD5. These operations can be outsourced to an external chip for computation and are the target of our optimization.

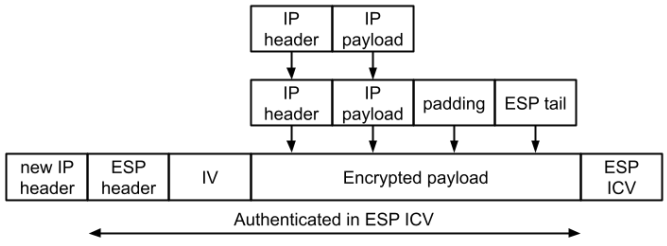


Fig. 2. IPsec ESP tunnel mode overview.

If we encapsulate all packets with IPsec, as shown in Fig. 3, the longjump approach achieves the highest performance, reaching 100k packets/sec. In comparison, the non-async and context modes achieved throughputs of 76k and 48k packets/sec, respectively.

We examined the impact on performance when decreasing the percentage of packets requiring IPSec while still forwarding the others without any extra processing. In Fig. 3, we can observe how the performance changes when only 50% or 33% of the packets are encapsulated by IPsec. As anticipated, there is a substantial performance improvement with both methods as the computation-intensive packet rate decreases. Due to the significant overhead of the context method and the relatively short complexity of the IPsec calculation, the context-based approach is slightly slower than the non-async case. In contrast, our solution clearly outperforms both of them.

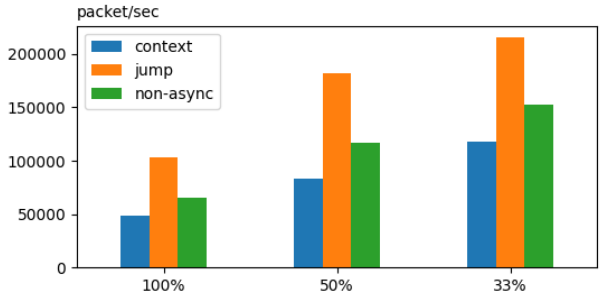


Fig. 3. Throughputs of different approaches on changing the percentage of IPSec packets in the traffic mix.

When we decrease the number of packets that require external processing, we can observe the outcomes presented in Fig. 4 and Fig. 5. The longjump approach shows the superior performance when the number of packets requiring cryptography operations is higher and can be optimized by asynchronous processing. As the percentage of these tasks decreases, the performance differences between the approaches become less significant, and they tend to converge as the tasks become simple packet forwards. In such cases, the approaches do not show any significant performance differences. The context-based approach cannot outperform the synchronous method in this case, and the article proposing it also indicates that this method is suitable for external tasks requiring longer run-times.

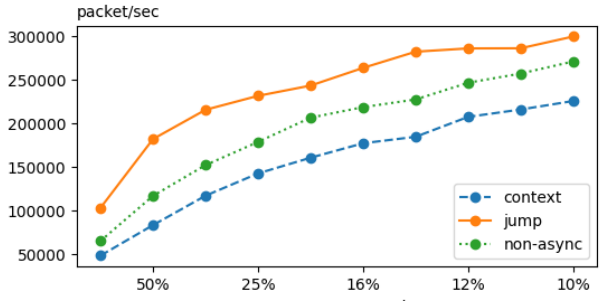


Fig. 4. Throughput of different approaches while changing the ratio of IPSec packets in the traffic mix from 100% to 10%.

We measured with different packet sizes, multiple cores, and burst sizes there was not any significant difference in the

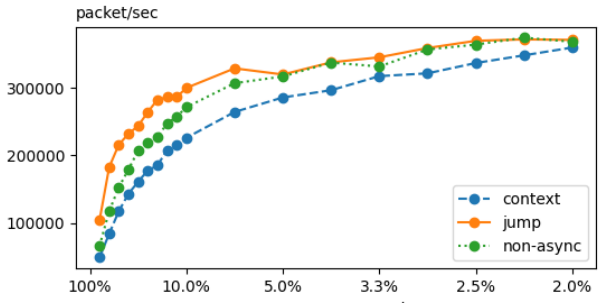


Fig. 5. Throughput of different approaches while changing the ratio of IPSec packets in the traffic mix from 100% to 2%.

results with correctly configured parameters (for example we need at least burst size 16).

B. Simple encryption

The study involved testing the performance of our new approach using simple symmetric encryption, such as AES. The results, shown in Fig. 6, reveal a narrower gap between our approach and the non-async case, but the longjump approach still achieves better performance overall if we encpryt enough packets. With the shorter external task, the context method incurs a higher overhead that is roughly three times worse than the non-async case. Similarly to the IPsec encapsulation experiment, as the percentage of encrypted packets decreases, the performance of the three approaches becomes more similar, with the longjump and non-async cases essentially performing identically at a 20% encryption rate.

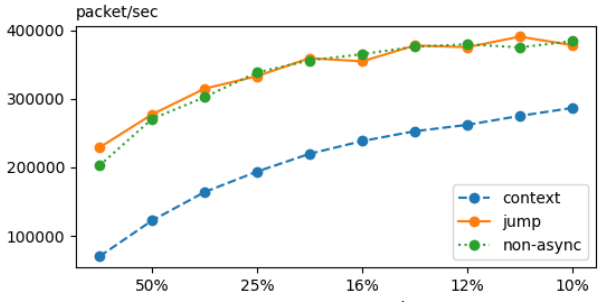


Fig. 6. Throughput for simple encryption with different approaches, varying the ratio of IPSec packets in the traffic mix from 100% to 10%.

C. Other external functions

In order to obtain a more comprehensive understanding of the performance benefits of the new approach, we conducted further investigations into the performance of custom external functions. To simulate a custom external function, we define the runtime of the function in cycles. Although we assume that all of the external functions have the same runtime, the large number of function runs allows us to disregard any differences and consider the mean runtime as the runtime for each function.

The experiment involved creating a program that simulates external hardware on a dedicated CPU core. When a task arrives, the program waits for the desired amount of time and then returns the data without making any changes.

In Figure 7, we can see a performance comparison of various external functions with different levels of complexity. The X-axis represents the cycles required for the operation, while the Y-axis shows the packet per second rate. When an external function requires a small number of cycles (e.g., 1000-2000), the overhead of reparsing the packet may outweigh the benefits of parallel processing, and asynchronous processing may not be necessary. The plots demonstrate that the non-blocking approach offers better performance for extremely simple functions. However, as the complexity of the function increases, the advantage of the longjump method becomes more pronounced.

It is observed in our simulation setup that for external functions with higher runtimes, the context approach outperforms the non-async mode, whereas the longjump approach performs better overall.

D. On stronger configuration

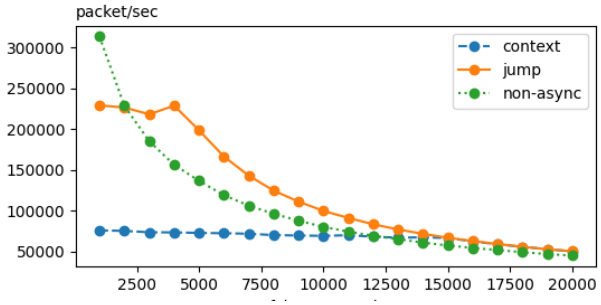
In a prior study by Laki et al. [5], they conducted performance measurements of the context-based approach on more powerful machines. To ensure comparability, we replicated their experiments in this section using machines equipped with an AMD Ryzen Threadripper 1900X CPU (8 cores/16 threads, 3.8 GHz) and 128 GB of RAM, along with a 10 Gbps NIC (Intel 82599ES). Both the packet generator and switch in our setup have this configuration and are directly connected to each other.

The results of the IPsec throughput measurement on this configuration are shown in Fig. 8. It can be observed that the context approach had slightly better performance than the synchronous solution over the entire investigated interval. On the other hand, the longjump approach achieved a significant speedup, as it was able to process over 930k packets per second when encapsulating all packets, while the context and synchronous solutions only processed about 320k packets per second. When encapsulating only 10% of the packets, the longjump approach still achieved more than a 12% speedup.

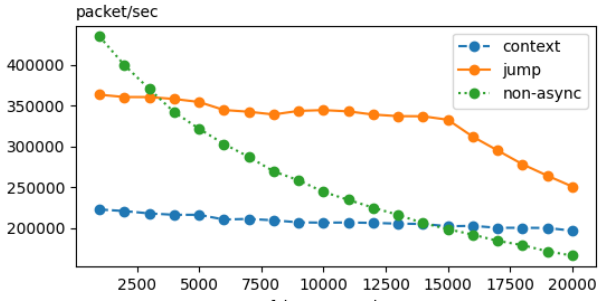
Regardless of the run-time of the custom external functions applied to all packets, the longjump approach provides higher throughput. Even if the function only requires 1000 cycles, the longjump approach performs better. However, if external functions are applied to only 10% of the packets, the competition is more balanced, as depicted in Fig. 9. The longjump approach, in general, surpasses the other methods in this scenario, and its performance improves further with more complex external functions.

V. Conclusion

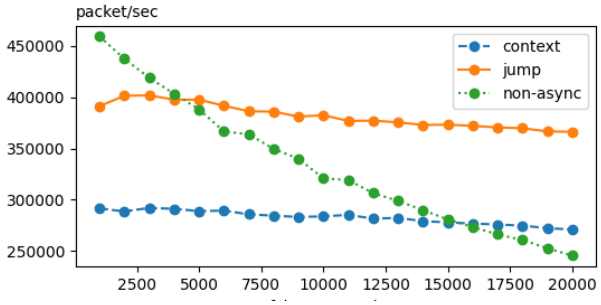
This paper introduces a new method for implementing asynchronous processing in P4 programs that utilizes the longjump technique, allowing for more efficient and flexible



(a) Running custom external function for all packets.



(b) Running custom external function for 20% of packets.



(c) Running custom external function for 10% of packets.

Fig. 7. Throughput for different external function complexities. X-axis shows the CPU cycles required by the external function.

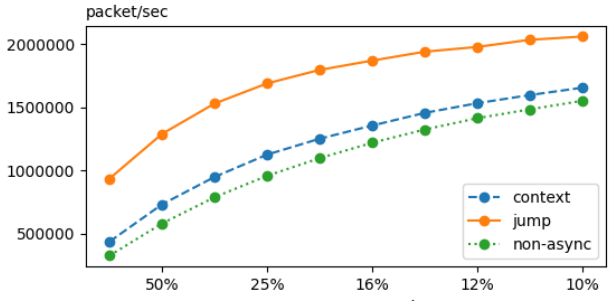


Fig. 8. Throughput for different approaches while varying the percentage of IPSec packets in the traffic mix on a high-speed server.

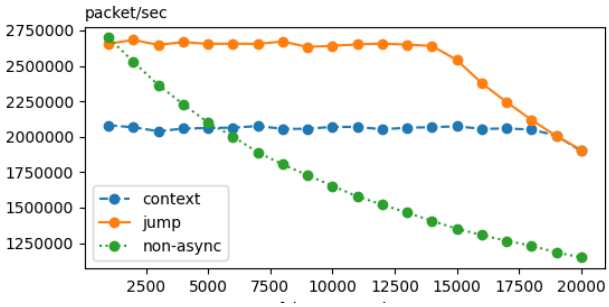


Fig. 9. Throughput for custom external functions with variable computational complexity (CPU cycles on x-axis). Extern was only applied on 10% of packets in the traffic mix.

use of data plane functions compared to the current context-based approach that is suitable only for longer external tasks. Through various experiments evaluating the performance in real-world scenarios, the study demonstrates that the longjump method outperforms both the non-async and context-based approaches for external tasks that require a minimum of 1k-2k clock cycles to complete. The method maintains high performance even when only a small percentage of packets require external processing, making it a practical option for most use cases.

Acknowledgement

P. Vörös thanks the support received from the ÚNKP-22-4 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund. S. Laki thanks the support of NRDI Office (NKFIH, FK_21 138949).

References

[1] M. Karakus and A. Durresi, "A survey: Control plane scalability issues and approaches in software-defined networking (sdn)," *Computer Networks*, vol. 112, pp. 279–293, 2017.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[3] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki, "T4p4s: A target-independent compiler for protocol-independent packet processors," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–8.

[4] F. Hauser, M. Häberle, M. Schmidt, and M. Menth, "P4-ipsec: Site-to-site and host-to-site vpn with ipsec in p4-based sdn," *IEEE Access*, vol. 8, pp. 139 567–139 586, 2020.

[5] S. Laki, D. Horpácsi, P. Voros, M. Tejfel, P. Hudoba, G. Pongracz, and L. Molnar, "The price for asynchronous execution of extern functions in programmable software data planes," in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. IEEE, 2020, pp. 23–28.

[6] D. Horpácsi, S. Laki, P. Vörös, M. Tejfel, G. Pongrácz, and L. Molnár, "Asynchronous extern functions in programmable software data planes," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019, pp. 1–2.

[7] S. Laki, R. Stoyanov, D. Kis, R. Soulé, P. Vörös, and N. Zilberman, "P4pi: P4 on raspberry pi for networking education," *ACM SIGCOMM Computer Communication Review*, vol. 51, no. 3, pp. 17–21, 2021.

[8] P4ELTE, "t4p4s: Retargetable compiler for the p4 language," https://github.com/P4ELTE/t4p4s/, Accessed: 2023-03-18.