The Price for Asynchronous Execution of Extern Functions in Programmable Software Data Planes

Sándor Laki, Dániel Horpácsi, Péter Vörös
Máté Tejfel, Péter Hudoba
Eötvös Loránd University
Budapest, Hungary
lakis@elte.hu

Gergely Pongrácz,
László Molnár
Ericsson Research
Budapest, Hungary
gergely.pongracz@ericsson.com

***Abstract*—Target-independent packet processing languages such as P4 support diverse hardware and software targets by generalizing over the set of primitive operations available on the target. Architecture models declare extern functions through which functionalities of the underlying target can be accessed. Though they can be invoked at any location in the packet processing pipeline, their implementation details are opaque to the data plane program. In P4, the language specification does not define whether the extern function calls are synchronous or asynchronous control statements — supposedly synchronous by default. However, there are use cases when the asynchronous invocation makes more sense, letting the main thread keep processing packets while the extern operation is being performed by a dedicated resource (dedicated thread, CPU core or GPU) or an accelerator device (cryptographic co-processor or accelerator card). This paper examines how asynchronous extern function calls can be implemented in high-performance software data planes defined in P4, what overheads must be taken into account and which factors affect the price we have to pay for the asynchronous invocation. Our evaluation reveals the trade-off between the packet processing performance and the ratio of the computational cost of the extern function to the overhead caused by the asynchronous execution model.**

I. Introduction

Software Defined Networking (SDN) [1] promises to provide computer networks with high degree of flexibility and scalability by decoupling data and control planes and introducing programming abstractions into both layers. Although control plane programmability has quite a long history in the literature, problems of programmable and portable data planes have started gaining notable attention in the recent years [2]. As a promising solution, specific programming languages have emerged, enabling network developers to describe the entire packet processing in a protocol-independent way at a high abstraction level.

Among the several language proposals, P4 [2] has gained the strongest community support, strengthened by members from both industry and academia. It is a target- and protocol-independent packet processing language which enables high-level description of packet handling algorithms. P4 programs can run as software on general-purpose processors or can control a dedicated network hardware: the language has several software implementations, it obtained some hardware implementations for NetFPGA [3] and SmartNICs, as well as it has its custom-designed set of ASICs, Tofino, developed by Barefoot Networks.

978-1-7281-5127-4/20/$31.00 ©2020 IEEE

This paper advances software implementations of P4, in particular, moving towards better exploitation of computational resources and effective integration of hardware accelerators into the P4-based packet processing pipeline. Software data planes play a key role in modern telecommunication systems, insomuch that in data centers most packets passing through virtual machines are processed by software switches. The number of hypervisor software switches in a typical data center may exceed the number of physical switches. With the advent of Network Function Virtualization (NFV), the network functions traditionally implemented by dedicated hardware are now realized as software components, promising higher flexibility and better scalability. One of the key advantages of software data planes is that they can be upgraded and scaled up by running multiple instances of the same switch program more easily compared to state-of-the-art hardware solutions [4].

Server computers where software data planes run can be equipped with hardware accelerators (e.g., Intel QAT, AMD CCP, CAVIUM Octeon) and other computational resources (such as GPUs, TPUs or other application-specific processors), which can take their part in the packet processing pipeline by offloading specific tasks to them. Also, for better usage of limited CPU resources, a partition of CPU cores can be dedicated to execute specific external functions like encryption/decryption, compression/decompression, running an artificial neural network on the GPU or any complex functions that cannot be described in P4. Such pipeline elements, or external functions, may appear in the middle of the control flow, therefore efficient offloading of these external functions requires the ability of asynchronous function invocation.

In this paper, we introduce a possible implementation of asynchronous external functions in programmable software data planes and extend our prior work [5] by a thorough evaluation showing that the decision of which execution model (synchronous or asynchronous) performs better depends on several factors including the overhead caused by the asynchronous execution, the computational cost of extern function and the composition of the network traffic.

Authorized licensed use limited to: Eotvos Lorand University Budapest. Downloaded on October 27,2022 at 11:03:06 UTC from IEEE Xplore. Restrictions apply.

The remaining part of the paper is organized as follows. After elaborating the problem statement in Section II, we define a solution and present its proof of concept implementation in the software data plane generated by our open-source P4 compiler called T4P4S [6] in Section III. In Section IV, a benchmark P4 pipeline is presented, followed by the evaluation of its performance with the proof of concept implementation. Section V discusses the limitations and future work, and finally, Section VI concludes the results.

II. Asynchronous Extern Functions

The P4 language handles the diversity of targets by making itself extensible via so-called extern objects and extern functions. P4 externs represent functionality in the architecture model that is implemented by the given target (either in software or hardware). Depending on the nature of the functionality, invocations of the extern function may be implemented to be synchronous or asynchronous:

1) When the thread invoking the extern gets blocked for the function execution and waits for the result, regardless of whether the function is running in a separate thread, the invocation is *synchronous*.
2) If the extern function is executed on a dedicated resource (e.g., in a hardware accelerator card, co-processor or just a dedicated thread) in a separate context and the packet processing thread is not blocked, then the invocation is *asynchronous*. In this case, the packet processing thread can keep handling other packets while the operation is being performed. This option pays off if the operation is complex enough making it worth handed over to a dedicated resource despite the costs of the transmission and context switch.

Suppose that a P4 control invokes an architecture-provided encryption functionality or any other complex operations. For example, Langlet et al. in [7] periodically applies an artificial neural network as an extern function on features collected by the data plane to detect network anomalies. In these cases, the function execution definitely takes notable CPU time, in which it should not block the fast path. In this case the asynchronous invocation is a good choice.

When executing extern functions asynchronously, the control flow exits the pipeline, the function gets processed by a separate unit, and on completion the control flow is directed back to the point where the extern call happened. From the packet handling point of view, the processing block is suspended at the extern call and is resumed after the function's return. Figure 1 depicts the workflow in an abstract way. The packet processing pipeline contains an asynchronous extern call, where asynchronous execution is needed. The pipeline starts with a parser and a control block (control-a — e.g., ingress block), in the middle there is an asynchronous extern call, and then another control block has to be executed. When reaching the extern call, the packet context is saved and the packet with the context information is forwarded to an input queue of the dedicated processing unit (different thread or hardware unit). After the extern call has returned, the packet is

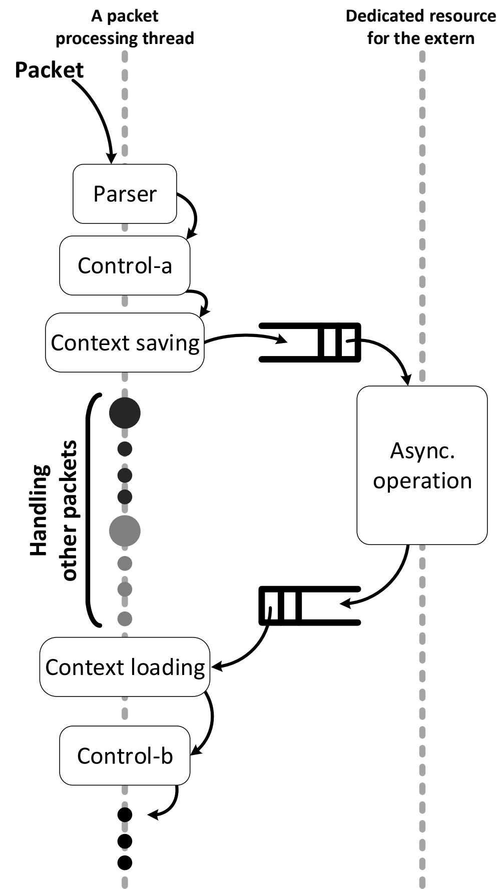

Fig. 1. Packet processing with an asynchronous extern call

forwarded back to the processing thread with its packet context through a buffer. The context is restored for the packet and the execution of the control block (control-b) is continued right after the extern call. Note that in many cases before context saving or after context loading additional preparation steps may be needed, e.g., preparing the packet representation for encryption or parsing additional headers after decryption.

At present, P4-generated software switches tend to lack of support for asynchronous extern functions, despite that fact that this is essential when it comes to offloading specific tasks to hardware accelerators. We have designed and implemented a method that allows P4 compilers to emit software switches that employ asynchronous calls.

Packet contexts

Other packets should not affect the context (local variables, metadata, etc.) of the offloaded, asynchronously processed packet. In fact, all concurrently processed packets need a separate context. Our method provides isolation and concurrent execution of packet handlers by making both the packet processing loop (main loop) and the packet handler function coroutines. Coroutines are functions whose execution can be suspended and resumed later on, and they allow us to run our packet handlers in manually scheduled lightweight threads.

Authorized licensed use limited to: Eotvos Lorand University Budapest. Downloaded on October 27,2022 at 11:03:06 UTC from IEEE Xplore. Restrictions apply.

By using coroutines, a single OS thread can run multiple computations concurrently. There is no free lunch: this solution needs to manage context saving and switching.

a) Parsed representation.: The packet context, along with a number of state variables, contains information about the header structure of the packet. This is called the parsed representation. Some extern operations may leave this intact by only affecting the payload of the packet, but others may change the header structure and therefore invalidate the parsed representation. Somewhat similarly, extern functions may operate on the raw packet as a byte array, meaning that the packet has to be deparsed before the operation. In general, some type of deparsing and parsing (serialization and deserialization) is needed before and after the extern operation. Our solution assumes that a pair of deparser and parser implementation is supplied with the extern function call to specify how serialization takes place.

b) Context and recirculation.: Typically, extern functions implement operations that are part of the pipeline, the context of the extern call has to be restored completely. On the other hand, there are some exceptions as well: in our case study, we examine encryption and decryption functionality. The latter very likely happens in the beginning of the pipeline, meaning that its context is practically empty. In such cases, when the context is only supposed to hold the control flow location, it may be pointless to use context saving and restoration, it makes more sense to simply recirculate the packet to the very beginning of the pipeline following the asynchronous operation.

III. Implementation

In order to demonstrate the proposed concept, we have modified our DPDK-based open-source P4 compiler and software switch, called T4P4S [6], to provide experimental support for asynchronous invocation of extern functions in P4 programs. With the modifications, the packet processing threads can handle multiple packets concurrently, resulting in seamless and effective integration of computation-heavy (or resource-heavy) extern functions into the pipeline. Concurrent packet processing within threads is achieved by employing asynchronous function execution by turning packet handlers into coroutines. Note that we plan to make the source code available in Github.

In the C program emitted by the T4P4S compiler, the packet processing pipeline (including both the ingress and egress controls) is implemented by a single packet handling function. Each CPU core runs a loop to execute this handler on each packet to be processed. Prior to the implementation of asynchronous externs, all functions blocked the execution of the pipeline; thus, they blocked the entire thread and the CPU core itself, preventing it from handling other requests while performing the extern function. In order to tackle this, we implemented the packet handling function to be able to spark asynchronous execution of particular extern functions in terms of coroutines.

Coroutines

Coroutines allow us to suspend and resume packet processing at asynchronous extern calls. Languages that implement coroutines usually come with two primitives *async* and *await*, which provide the vocabulary for turning functions into coroutines as well as for suspending their execution. The C language does not support coroutines by design, but there are libraries that implement this feature. In our C programs, we realize coroutines with the ucontext (user thread context) module found in the C standard library for System V[1].

Concurrent execution with packet handler coroutines needs manual scheduling, i.e. manual context switching between the main loop and the packet handlers. In the ucontext library there is an operation called swapcontext: it takes two coroutine contexts, suspends the one and resumes the other. Coroutine contexts are made with two primitives: getcontext and makecontext.

a) Context switching: When a packet arrives, the main loop creates and initializes a packet (sub)context (a packet handler coroutine instance), and by swapping to the newly created context, suspends the main loop and starts the packet handler. Thus, the thread starts executing the pipeline for the packet. When an asynchronous extern function gets invoked in the pipeline, the thread swaps back to the main loop and starts processing other packets while the extern function is being performed. Meanwhile, it monitors a dedicated queue for the result of the extern operation. When the result is available, another context switch suspends the main loop and resumes the pipeline exactly where it has been suspended by the extern call. When the pipeline is completed and the packet handler function reaches its end, a context switch brings the control back to the main loop.

b) Representation switching: The inner representation of packets in our generated switch is a structure that keeps track of the packet data and its metadata, maintains pointers to the headers and the fields in the byte array, and also contains some descriptors that guide parsing and deparsing. In the DPDK prototype implementation, when a packet is sent for an asynchronous operation, this inner representation is serialized to a memory buffer. This buffer, besides including the packet data, contains the overall size of the packet, the type and the arguments of the asynchronous operation, as well as it remembers the identifier of the packet context. The serialized data is put into a buffer for asynchronous operations. Once a burst of such operations is ready for processing, they are enqueued to the extern's own queue in a burst.

Similarly, when the asynchronous operation is completed, the memory buffers are dequeued from the extern's result queue in a burst. Then, deserialization extracts the altered packet data, the packet size and the context identifier. An inner representation gets rebuilt from the extracted information, and the parser control is invoked to rebuild the header descriptions.

[1] In the future, this may be replaced by another implementation as ucontext became deprecated in POSIX 6 and was removed in POSIX edition 7.

Authorized licensed use limited to: Eotvos Lorand University Budapest. Downloaded on October 27,2022 at 11:03:06 UTC from IEEE Xplore. Restrictions apply.

```
apply {
  smac.apply();
  dmac.apply();
  if (meta.do_extern == 1) {
    @async( deparser = "MyDeparser",
            parser   = "MyParser")
    {
        extern_function();
    }
  }
}
```

Fig. 2. The control block used in the evaluation where the extern_function is either encryption or a dummy function emulating operations with various computational costs.

c) Metadata preservation: In the prototype implementation, we assume that the metadata is not required for the extern operation, the extern function only affects the packet data. On the other hand, metadata has to be preserved and made available when the packet returns from the extern function. One could simply bring the metadata headers along with the packet data towards the extern function, but copying unneeded data (possibly to a separate hardware component) would take undue data transmission. Our solution is more in line with user contexts: we simply copy the otherwise heap-stored data into a local variable and let the context switch handle metadata restoration.

d) Context pools: As we discussed it already, asynchronous execution takes place in user contexts, rather than in threads. Since C does not support contexts on the language level, allocation and deallocation of contexts has to be carried out by the program itself. We maintain a dedicated memory pool to provide space for contexts.

IV. Evaluation

This section provides the preliminary results on the performance evaluation of the proposed implementation, focusing on three main questions: 1) What is the overhead of asynchronous execution of an extern function? 2) What is the contribution of context creation and context switching in the packet forwarding performance? 3) How does the proposed method scale with the available CPU cores under various traffic load?

The measurements have been carried out in our local testbed consisting of two identical nodes (AMD Ryzen Threadripper 1900X 8C/16T 3.8 GHz, 128 GB RAM): a traffic generator node using DPDK's Pktgen tool to generate test traffic and a P4 switch node executing the proposed implementation. Each one is equipped with a 1 Gbps NIC for management purposes and a dual port 10 Gbps NIC (Intel 82599ES) for the measurements. On both machines, the two 10 Gbps interfaces are used with Intel DPDK drivers.

For the performance analysis we have implemented a benchmark P4 pipeline that is based on a simple L2-forwarding program consisting of two exact tables, `smac` and `dmac` filled with the source and the destination MAC addresses of flows in the generated test traffic. Figure 2 presents the apply block of the ingress control of the benchmark P4 program. For technical reasons each asynchronous function has its own annotated internal block. Annotations are used to define which deparsing method shall be applied before and which parsing method after the execution of the asynchronous extern. In our example, the original parse and deparse methods of the L2 forwarding can be applied by both asynchronous calls. Note that some methods like encryption and decryption may require the serialization of the packet content in advance, but other extern functions may work without these steps, thus deparse and parse parameters can be left undefined.

The extern operation is called after applying the `dmac` table. We use a dummy extern function that can emulate operations with arbitrary computational costs expressed in CPU clock cycles. The extern function can be applied to all the packets flowing through our switch program or only to a portion of the traffic. In this paper, 4 extern percentage settings have been examined: 10%, 20%, 50% and 100% of the packets trigger the call of the extern function at the end of the pipeline (`meta.do_extern` is set in Fig. 2), while the remaining packets end the pipeline after applying table `dmac`. Note that this selection is applied on the packets received by one of the packet processing threads and are not affected by the packets dropped outside the software switch (e.g., by the NIC). All the settings have been evaluated with both asynchronous and synchronous (without @async annotation) executions. In both cases, the extern function is running on a separate CPU core emulating the dedicated resource that is responsible for the specific extern operation.

Fig. 3 depicts the examined scenarios using the dummy extern function with variable computational costs from 50 CPU clock cycles to 10000. Note that the 50 and 10000 CPU clock cycles correspond to approx. $13ns$ and $2.6\mu s$ delays, resp. The test traffic has been generated at line rate, consisting of 10 flows with packets of size 64 bytes. The figure shows the single CPU core performance where the main packet processing pipeline runs on a single isolated CPU core and another dedicated core is used for serving the extern calls. One can observe that as the extern call percentage decreases, not surprisingly the resulted packet forwarding rate increases, since less packets have to go through the costly extern operation. Note that the single core forwarding performance of the benchmark pipeline without the extern call (0% extern call percentage) is approx. 11 MPPS. One can also see that the most important factor in the decision of which execution model performs better is the computational cost of the extern operation. Synchronous execution clearly outperforms asynchronous one if the computation cost of the extern function is small, e.g., up to 2000-3000 clock cycles in our case. As a given complexity in the extern function is reached, the overhead caused by the asynchronous execution can be compensated and thus higher throughput can be reached. The right plot in Fig. 3 shows that the performance of asynchronous execution crosses the synchronous ones at 2000 clock cycles and after this point the asynchronous execution is better since the extern

Authorized licensed use limited to: Eotvos Lorand University Budapest. Downloaded on October 27,2022 at 11:03:06 UTC from IEEE Xplore. Restrictions apply.

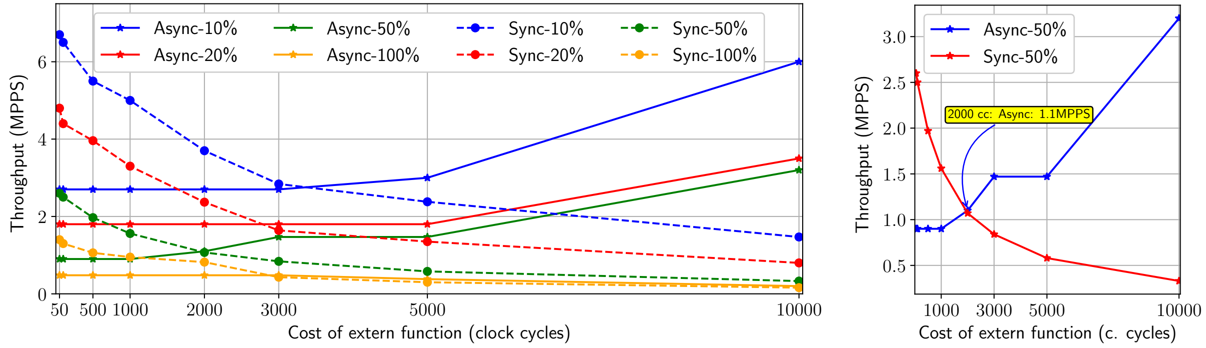

Fig. 3. Performance of synchronous and asynchronous execution models for extern functions with various computational costs. The extern operation is applied to 10%, 20%, 50% or 100% of the incoming traffic.

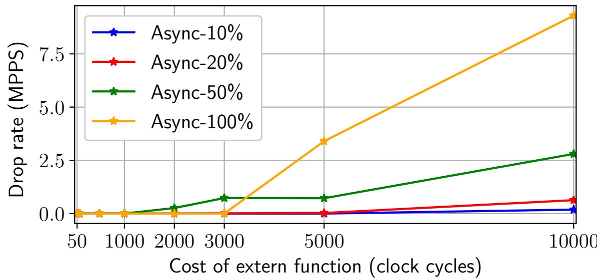

Fig. 4. Drop rate at the extern buffer. The buffer can store 1000 packets. If it is exceeded, the packet is dropped without context creation.

operation becomes a bottleneck that in asynchronous case only affects the thread executing the extern function instead of the main packet processing thread. Though our proof of concept prototype based on ucontext library is not optimal, we expect similar phenomena with more optimal implementations where asynchronous way could cross the synchronous one at smaller computational costs.

Fig. 4 shows packet drops caused by the full extern buffer. In our implementation the extern buffer can store at most 1000 packets, if it is filled packets are dropped at the extern call. For small number of clock cycles, the extern buffer causes almost zero packet drop, but above 2000 CPU cycles (approx. $0.5\mu s$) in the 50% and 100% cases the drop rate increases, indicating that packets started accumulating in the buffer. Comparing the cases of 50% and 100% extern percentages drop rate starts increasing earlier with the former — all the packets call the extern operation and thus the overhead of context creation are very high. The time needed for processing a single packet is significantly increased, and thus packets are dropped outside the switch program by the NIC.

In Fig. 5 we show how the performance scales with the number of CPU cores. Note that the extern function still runs on a single dedicated core and only the number of main packet processing threads is changed. In this scenario, we consider 10% extern percentage and three extern functions: 1) Encryption that uses the OpenSSL driver of DPDK (instead of our dummy extern function) and 2) dummy extern function with computational cost of 3000 and 3) 10000 clock cycles. One can observe that both synchronous and asynchronous executions scales well with the number of CPU cores and in case of encryption the overhead caused by the extern calls is much larger than the cost of the crypto operation.

Fig. 6 depicts a scenario where the software switch running the benchmark pipeline is not flooded with the test traffic. The sending rate in the traffic generator is set to the maximum where the observed packet drop rate is almost zero (similarly to RFC2544 [8]). The dummy extern with a 5000 clock cycles (approx. $1.3\mu s$) cost is applied with different number of CPU cores and extern percentages. In accordance with Fig. 3 the asynchronous execution with this computational complexity of the extern operation clearly outperforms the synchronous one. The numbers are in accordance with our previous ”flooding-based” measurements. One can also observe that the scalability with the number of CPU cores is also affected by the traffic's extern percentage which is caused by the single dedicated resource for the extern execution. For at least 20% extern ratio the performance of synchronous operation with two CPU cores can be approached by the asynchronous results with a single CPU core. Note that the behaviour of the synchronous execution with two CPU cores are very similar to the single core asynchronous case. As the percentage increases the single resource handling the extern calls starts limiting the overall performance.

V. Discussion

Though the proposed method makes the asynchronous execution of P4 extern functions possible, there are a number of open issues not or only partially managed in our solution.

1) Some asynchronous operations e.g., encryption, compression may require the partial serialization of packet headers while reparsing may be needed for others like decryption or decompression. Our benchmark P4 program uses different internal blocks for every call of asynchronous extern functions where blocks are annotated with the execution mode and further parameters. Using this technique all the necessary information needed for the implementation of asynchronous

Authorized licensed use limited to: Eotvos Lorand University Budapest. Downloaded on October 27,2022 at 11:03:06 UTC from IEEE Xplore. Restrictions apply.

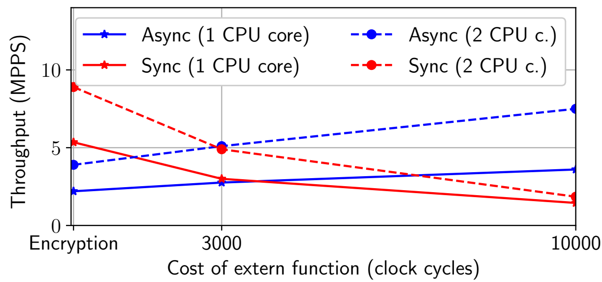

Fig. 5. Scaling with number of CPU cores. Extern function is applied to 10% of incoming packets.

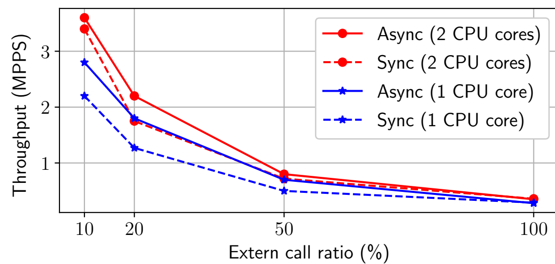

Fig. 6. Maximum achievable packet rate with almost zero drop rate (similarly to RFC2544). The extern function takes 5000 clock cycles (approx. $1.3\mu s$).

calls can be provided to the P4 compiler, but language support for asynchronous operations including hardware acceleration in P4 would be much cleaner.

2) In our early prototype context handling is based on the System V Context C (ucontext) library that saves or reloads the whole stack of the generated switch program at given checkpoints. For demonstrating the concept it works, but it results in too large overhead reducing the packet processing performance. As shown in Section IV, the primacy of asynchronous execution mostly determined by the ratio of the overhead caused by the context management and the computational complexity of the extern function. For extern functions with very low costs, asynchronous execution may results in about 33%-50% percent of the achievable maximum throughput of the synchronous operation, depending on the traffic mix. Our measurements has shown that context creation has the biggest impact on the performance and context switching has much lower cost. One of the reasons is that the packet context in the generated switch program is much larger than what is really needed. According to the P4 code, the packet context can be defined more precisely, limiting its size and thus context handling can be fasten by many factors.

3) The proposed approach can be extended in a straightforward way to support not only asynchronous extern functions but asynchronous execution of P4 instructions or blocks of instructions (e.g. complete controls). This could be important if distribution of computational resources needs to be optimized.

4) Our analysis envisions that if the cost of an extern operation can be predicted, the price of using asynchronous or synchronous execution and the effect on the overall performance may also be modelled. Then an automated decision on the execution model to be used could be made to optimize the operation of future software data planes.

VI. Conclusion

Efficient execution of computationally expensive functions requires the ability of asynchronous function invocation. This paper presents a solution to support asynchronous extern functions by using context saving and switching. The method is implemented using the DPDK-based T4P4S compiler and software switch. Illustrating the applicability of the proposed approach a benchmark P4 program has been created and measurements are presented to proof the concept and analyze its advantages and disadvantages. Our evaluation shows that the decision of which execution model (asynchronous or synchronous) performs better relies on different factors. The extra overhead caused by the asynchronous execution (e.g., context creation and switching), the computational complexity of the extern operation and the ratio of these to each other play the most important roles in the decision. The paper also discusses the main open issues implying room for possible future work and improvements.

Acknowledgement

The authors thank the support of Ericsson Hungary Ltd. and the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013). S. Laki thanks the support of the grant: Project no. ED_18-1-2019-0030 (Application-specific highly reliable IT solutions) has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme funding scheme.

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] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, “Netfpga–an open platform for gigabit-rate network switching and routing,” in *2007 IEEE Int. Conference on Microelectronic Systems Education (MSE’07)*. IEEE, 2007, pp. 160–161.

[4] T. Lévai, G. Pongrácz, P. Megyesi, P. Vörös, S. Laki, F. Németh, and G. Rétvári, “The price for programmability in the software data plane: The vendor perspective,” *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 12, pp. 2621–2630, Dec 2018.

[5] 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)*, Sep. 2019, pp. 1–2.

[6] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki, “T4p4s: A target-independent compiler for protocol-independent packet processors,” in *Proceedings of the International Conference on High Performance Switching and Routing 2018 (HPSR’18)*. IEEE, 06 2018.

[7] J. Langlet, A. Kassler, and D. Bhamare, “Towards neural network inference on programmable switches,” *Talk at EUROP4 Workshop 2019, Cambridge, UK*, 2019.

[8] J. M. S. Bradner, “Benchmarking methodology for network interconnect devices,” Internet Requests for Comments, RFC Editor, RFC 2544, 3 1999. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2544.txt

Authorized licensed use limited to: Eotvos Lorand University Budapest. Downloaded on October 27,2022 at 11:03:06 UTC from IEEE Xplore. Restrictions apply.