Dynamic Data Plane Updates using Lua and libmoon

Manuel Simon, Sebastian Gallenmüller, and Georg Carle
Chair of Network Architectures and Services, Technical University of Munich, Germany
{simonm|gallenmu|carle}@net.in.tum.de

***Abstract*—Upcoming communication networks, such as 6G, require both high performance and reliability, while service updates typically introduce service downtimes. This study explores dynamic network function updates using libmoon, a DPDK-based high-performance packet processing framework. The approach enables seamless, on-the-fly updates of network functions. By leveraging LuaJIT, we profit from just-in-time (JIT) compilation, allowing for efficient per-flow function updates. Our evaluation demonstrates the feasibility of runtime re-programmability in network data planes. We show the induced latencies of runtime changes and examine cross-flow and cross-core influences. Moreover, we investigate the effects of JIT compilation and show the significance of JIT compilation for long-term performance.**

***Index Terms*—Dynamic Network Function, Data Plane Updates**

I. Introduction

Modern communication networks have to offer a reliable but high-performant connection. So-called network functions are software components that fulfill specific tasks in a network. Updates for maintaining or improving these network functions usually cause downtimes. We investigate software-based dynamic network functions to mitigate this problem. Dynamic updates allow for the continuation of the service without service degradation. Furthermore, it allows the service migration to other nodes by sharing and dynamically installing (parts) of the service. Going one step further, network operators can offer programmable data paths, allowing for customer-defined, per-flow packet processing. Dynamic programs can, e.g., be used for customized in-network computation or recovery failovers.

LuaJIT is a just-in-time (JIT) compiler for Lua programs. JIT compilation seems to be a promising candidate to realize such dynamic function updates, as the potentially costly optimizations can be performed on-the-fly. This paper describes and analyzes a prototype implementation of dynamic network functions in libmoon, a Lua-based DPDK wrapper. The implementation allows for the dynamic addition of new functionality defined by its source code piggybacked by a packet. The function is installed, i.e., changing the source code of the running network function, and JIT compiled. We show the influence of JIT compilation on the applicability of the approach in the short and long term and the impact on other flows and CPU cores.

II. Related Work

In previous work [1], we integrated dynamic eBPF processors into the P4 pipeline, following similar goals. eBPF can be pre-compiled using platform-independent byte code, allowing fast, dynamic updates. While it was not possible to exchange such programs using source code, it is possible for Lua. This study focuses on implementing dynamic network functions with Lua instead of eBPF. Furthermore, we look deeper into the cross-flow and cross-core dependencies of such updates and build a suitable testbed and experiment workflow.

Runtime reprogrammability has been investigated as capsule-based active networking [2]. Recently, there was work on dynamic updates in (P4) programmable switches. Das et al. [3] implemented an instruction set in P4 to modify packet processing during runtime. Xing et al. [4] built FlexCore, which enables a partial reconfiguration of data planes during runtime. Feng et al. [5] extended P4 for in-situ reprogrammability. Outside the P4 domain, Jeykumar et al. introduced "tiny packet programs (TTPs)" [6] that are specified by active packets and can execute a very restricted amount of instructions.

Using libmoon, we use a framework that is not restricted in its functionality and scales using multiple CPU cores.

III. Background & Implementation

This section describes our prototype implementation of dynamic network functions in libmoon and introduces the features used in Lua and LuaJIT.

A. Lua, LuaJIT, & libmoon

libmoon [7] is a Lua library that combines the advantages of DPDK and LuaJIT [8], providing high-performance packet processing and easy programmability. The libmoon project is a generalization of the MoonGen packet generator [9] and is also based on DPDK and LuaJIT. LuaJIT provides a JIT compiler for Lua, a scripting language, e.g., used for games. DPDK is a framework for high-performance user space software packet processing, bypassing the Linux kernel network stack and using poll mode drivers. libmoon uses JIT compilation to provide a high-level DPDK API, combining the high-performance with high-level language features, such as automatic memory management. Using Receive Side Scaling (RSS) libmoon can statically distribute different flows to different CPU cores, for parallel processing.

B. Prototype Implementation

Our implementation allows a dynamically defined function per flow, here defined by the source IP address. The address is used to look up the associated function in a hashtable. The

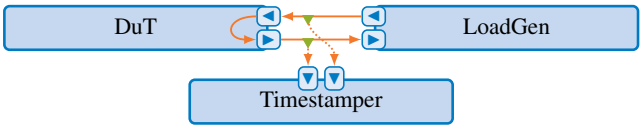

Figure 1: Measurement setup.

approach can easily be extended to other flow definitions, such as the 5-tuple. If no dynamic function has been set, a default function will be used, e.g., simply forwarding the packets.

New functionality can be installed by sending an *update packet* containing its Lua source code to the data plane adressing a pre-defined port. The source code is extracted from the update packet and converted to a callable Lua function using `loadstring()` and stored in the flow-function hashtable. This Lua built-in function allows the definition of new functions from source code returning a function pointer, similar to Python's `eval()` method. Before passing the source code, we extend it to correctly pass the arguments, i.e., the packet buffer and possible annotations for the JIT optimizations. The LuaJIT compiler will compile the function transparently, and the following packets of the same flow will automatically call the installed function. The function hash table is unsynchronized between multiple processing threads and CPU cores. However, the function is defined per flow, and RSS maps all packets of the same flow to the same CPU core.

IV. Setup

This section describes our experiment setup.

A. Topology

We use a three-host setup for evaluation (cf. Figure 1), that hardware timestamps all packets with a resolution of 12.5 ns [10]. All three hosts use an Intel Xeon D-1518 CPU @ 4×2.2 GHz, 32 GB of RAM and an Intel X552 NIC.

The load generator (LoadGen) utilizes MoonGen [9] to generate CBR traffic of two different flows, i.e., different source IP addresses, with a total rate of 200 Mbit/s and a packet size of 200 B. The Device-under-Test (DuT) runs the prototype implementation written for libmoon. The packets are processed individually without batching to determine per-packet costs. The DuT applies or changes the function and returns all packets to the LoadGen. Using passive optical splitters, all traffic is duplicated and mirrored to a third host, the Timestamper. The Timestamper timestamps and records all incoming packets. The recordings are used later to match and analyze the traffic and calculate the associated latencies.

B. Methodology

The LoadGen sends two types of packets: packets where the network function is applied (referred to as INC) and *update packets* that specify a new function that is installed (referred to as DYN). First, 50 k INC packets are sent from two different flows, where the default function is applied. Then, *one* DYN packet is sent, containing the source code of the dynamic function. Therefore, only one flow is affected by the dynamic function change. Afterward, another 200 k INC packets are sent from both flows in total, where the new, dynamic function is applied for the affected flow. The *other*, unaffected flow stays with the default forwarding function.

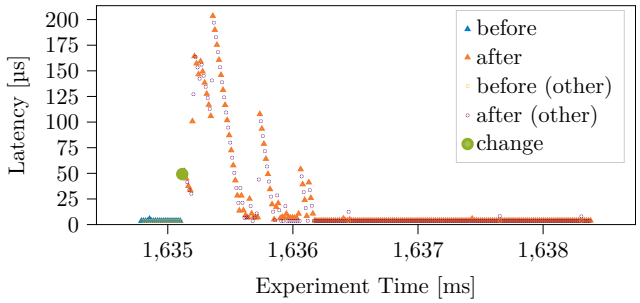

Figure 2: Dynamic program change (one task) (zoomed).

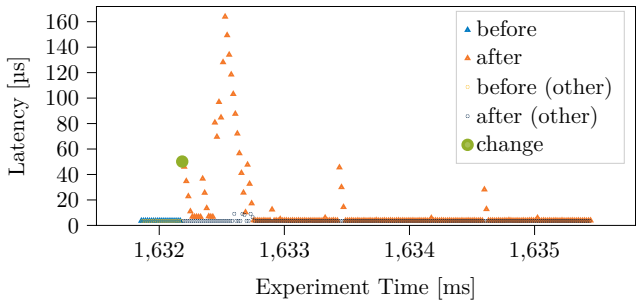

Figure 3: Dynamic program change (two tasks) (zoomed).

The experiments can potentially differ in the first-installed default function, the dynamic function sent and installed by the DYN packet, the number of tasks/threads, which handle the packets at the DuT, and the JIT parameters. *Here*, we analyze the basic overhead of changing functionality; therefore, we change the default function setting one constant in a packet towards a dynamic function setting another constant value. We perform this using one or two cores/threads while enabling or disabling the JIT compilation of Lua using the highest optimization level (cf. [8], similar to the `-O3` option of gcc).

V. Evaluation

This section describes the experiments that were performed and analyzes their results.

First, we investigate how changing the network function impacts the latency of the DYN packet and subsequent INC packets. Figures 2 and 3 show the latencies for 20 packets before and 200 packets after the change for the affected and the *other*, unaffected flow. When the packet processing is performed on a single core (cf. Figure 2), the change of the functionality affects *all* flows. In case the flows are split among several cores (cf. Figure 3), the other cores and the associated flows are unaffected. The median latency of the involved flow (also cf. Figure 6 in blue) increases slightly from 3.550 µs to 3.737 µs, while the other flow remains unchanged. The change itself introduces a latency of 50 150 µs, an overhead of 46 600 µs. However, no packet loss occurs in both experiments.

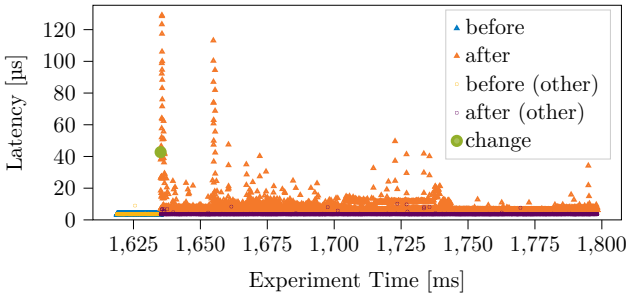

Figure 4: Dynamic program change (two tasks) (*no* JIT).

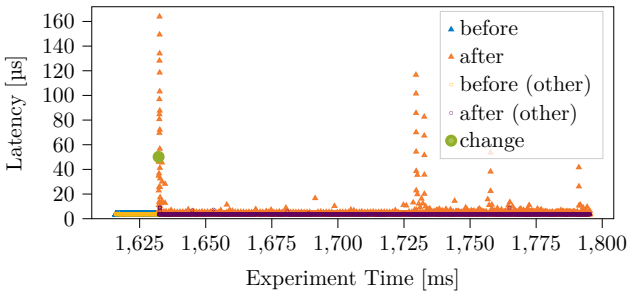

Figure 5: Dynamic program change (two tasks).

We further investigate the influence of the JIT compilation on the optimized performance of the changed, dynamic function and the impact on the change itself. Figures 4 and 5 depict 1 k INC packets before and 10 k packets after the change for two flows, respectively. The JIT compilation for the newly installed dynamic function is turned off in Figure 4 and activated in Figure 5. The variety and performance without JIT are, as expected, worse in the long term. In this case, Lua interprets the source code instead of using the JIT-compiled binary code. While the median latency before the change is 3.550 µs, it increases to 6.387 µs afterward. The other flow on the other core remains unchanged. Figure 6 additionally depicts the median latency of the first 5 k packets after the change. Enabling JIT compilation not only shows better performance in the long term but also in the short term. Nevertheless, the DYN packet latency itself is lower for the non-JIT than the JIT version (42 737 µs vs. 50 150 µs),

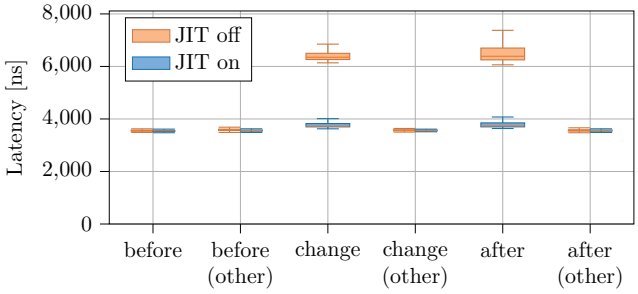

Figure 6: Latencies *before*, 5000 packets after the *change*, and there*after*).

probably due to the overhead of the JIT compilation itself. However, in both cases, there are latency spikes after the change, and the latency and its variance are increased. More experiments have to be conducted to get significant numbers.

VI. Discussion & Conclusion

This work investigated a prototype implementation of dynamic network function changes within libmoon. The results show that it is feasible to perform such changes, even for uncompiled source code. JIT compilation significantly reduces the overhead in the short and long term.

In future work, we will compare and analyze the influence of different optimization strategies supported by the JIT compiler in more detail. Furthermore, we will compare the technologies to other possible implementations, based on, e.g., P4 and eBPF [1], native eBPF/XDP implementations, or other (JIT-compiled) languages. A special remark lays on the interplay of the multiple CPU cores involved in the processing and the influence of changes on them and their caches. While this work only investigated the base overhead of changing functionality, different types of programs (e.g., computation or memory-heavy) may involve additional overhead. Moreover, the possibility of offloading such dynamic functions to SmartNICs, such as the Netronome Agilio and Nvidia Bluefield, may be investigated.

Acknowledgments

This work was supported by the EU's Horizon 2020 programme as part of the projects SLICES-PP (10107977) and GreenDIGIT (4101131207), by the German Federal Ministry of Education and Research (BMBF) under the projects 6G-life (16KISK002) and 6G-ANNA (16KISK107), and by the German Research Foundation (HyperNIC, CA595/13-1).

References

[1] M. Simon, H. Stubbe, S. Gallenmüller, and G. Carle, "Honey for the Ice Bear - Dynamic eBPF in P4," in *ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, 2024, p. 44–50.

[2] D. L. Tennenhouse and D. J. Wetherall, "Towards an Active Network Architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 26, no. 2, p. 5–17, apr 1996.

[3] R. Das and A. C. Snoeren, "Memory Management in ActiveRMT: Towards Runtime-Programmable Switches," in *ACM SIGCOMM 2023*, p. 1043–1059.

[4] J. Xing, K.-F. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, and A. Chen, "Runtime Programmable Switches," in *NSDI 2022*, pp. 651–665.

[5] Y. Feng, Z. Chen, H. Song, W. Xu, J. Li, Z. Zhang, T. Yun, Y. Wan, and B. Liu, "Enabling In-situ Programmability in Network Data Plane: From Architecture to Language," in *NSDI 2022*, pp. 635–649.

[6] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility," in *ACM SIGCOMM 2014*, p. 3–14.

[7] GitHub, "libmoon/libmoon," 2025, last accessed: 2025-02-16. [Online]. Available: https://github.com/libmoon/libmoon

[8] Mike Pall, "The LuaJIT Project," 2023, last accessed: 2025-02-16. [Online]. Available: http://luajit.org/

[9] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *IMC 2015*.

[10] Intel, "Intel Ethernet Controller X550 Datasheet rev 2.6," 2021, Last accessed: 2024-09-13. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/333369/intel-ethernet-controller-x550-datasheet.html