Proceedings of the 18th Linux Audio Conference (LAC-20), SCRIME, UniversitÃľ de Bordeaux, France, November 25–27, 2020

EXPRESS DATA PATH KERNEL OBJECTS FOR REAL-TIME AUDIO STREAMING OPTIMIZATION

Christoph Kuhr

Brühl, Germany
christoph.kuhr@web.de

Alexander Carôt

Anhalt University of Applied Sciences
Köthen, Germany
alexander.carot@hs-anhalt.de

ABSTRACT

Using a JACK media clock listener to synchronize JACK to an AVTP media clock talker results in performance issues when used with a raw Ethernet socket under Linux. The packet rate of a class A AVTP audio stream of 8 kHz triggers too many interrupts in the CPU. As a result a JACK audio cycle has only 125 μsec to process the audio data of all JACK clients. This restriction prevents such a system from real-time signal processing. The extended Berkley Packet Filter in combination with the express data path kernel features, that are integrated in the Linux kernel since version 4.8, are investigated. We could optimize the media clock synchronization by using a eBPF XDP program for pre processing of the stream packets. Our described solution is meant as an alternative to the usage of generic raw sockets.

1. INRODUCTION

Soundjack [1] is a real-time communication software using peer to peer connections, to connect up to five participants with each other. The targeted user group consists mostly of musicians. It was first published in 2006 [2]. The interaction with live music over the public Internet is very sensitive to latencies, both round trip as well as one-way. A rehearsal environment for conducted orchestras via the public Internet is the the ultimate goal for this research. Up to 60 musicians and a conductor shall be able to play together live.

Signal processing requirements make a server network mandatory, that connects up to 60 UDP streams to each other and mixes them. A single optimized processing server could handle the process of mixing this amount of concurrent UDP streams with reasonably low latency. Future research, however, shall investigate the application of immersive audio technologies in real-time. A single server would not be able to handle such computational load, since any filter calculation has to be done more than 60 times. Thus, a scalable server network provides the required processing power for a subset of the streams. The audio signals are routed between the signal processing applications via JACK [3]. JACK is a professional and open source audio server, that allows applications to share sample accurate audio data with each other. The servers need to share the processed audio data amongst each and have to be synchronized in time. For this purpose the AVB technology defined by IEEE standards (IEEE 802.1AS, 802.1Qat, 802.1Qav and 1722) is used. The AVB standards extend generic Ethernet networks with precise time synchronization, network resource reservation and bandwidth shaping. These properties avoid the Soundjack client streams from interfering with each other and also ensures the sample accurate synchronization of audio data across multiple servers.

This media clock synchronization of the multiple JACK instances on all servers, with the JACK AVB media clock listener (*avb-mcl*) backend was presented in [4]. Further investigations have shown that JACK is not able to keep the media clock in sync, if the local processing demand rises to the intended amount. The reason for this is the asynchronicity between the AVB AVTP packet rate for stream reservation class A traffic with a transmission interval of 125 μsec. At a sampling rate of 48 kHz, each AVTP packet contains 6 audio samples. The JACK sample buffer, however, always has a size of the power of 2 (e.g. $2^6 = 64$ samples), which 6 is not. Thus, with any sample buffer setting, multiple AVTP packets have to be received in a single JACK audio cycle. With 64 samples 11 AVTP packets are required. This means, that for any one of the eleven AVTP packets, the kernel has to allocate meta data and switch the process context to call the user space application. A JACK audio cycle for 64 samples, which requires 1.3334 $msec$ at 48 kHz to complete, is therefore interrupted any 125 μsec. But the situation is even worse, since the design of *avb-mcl* blocks until the next arrival of an AVTP packet. Consequently, it blocks 10 times and only leaves 125 μsec for the processing of an audio cycle overall. This is exactly the duration between the arrival of the 11th packet and the deadline of the audio cycle. This makes it nearly impossible to deploy *avb-mcl* in a productive environment.

A common solution to this problem is the outsourcing of the packet reception into a different thread. However, this would require synchronization of the threads and would introduce latency by locking or busy waiting. The achievement of the lowest possible kernel latency for this desired behavior with classical methods, would require to write a specific kernel module. This is a difficult task due to several reasons. Another possible solution has found its way into the Linux kernel in 2016, which we will explore in this paper: eXpress data path (XDP).

1.0.1. extended Berkley Packet Filters and eXpress Data Paths

Network traffic nowadays may easily require bandwidths, e.g. 100 *Gbps*, of a computer system's data bus and CPU that a generic software stack is unable to handle. Thus, it makes it hard to process packets within a reasonable reaction time. The reason for this limitation can be found in the allocation of meta data for billions of packets per second by the kernel. Not every packet, however, requires handling by the software stack. Use cases exist, that can be significantly sped up by preprocessing of packets inside the kernel. For most software developers this meant to write their own kernel modules, which is a very delicate and complex process. Three different strategies to accelerate and optimize network packet processing on a Linux computer exist:

- Kernel Bypassing
- Customized Kernel Module
- extended Berkley Packet Filter (eBPF) with eXpress Data Path (XDP)

Kernel Bypassing disables all features of the kernel. Several techniques exist that can be used for kernel bypassing. All of which, however, require dedicated network adapters. A customized kernel module requires a significant development effort. The source code of kernel modules for network adapters easily contains tens of thousands of lines of code, that are carefully tuned. Adding even small features may create unforseen development and debugging effort. Therefore, these two strategies are not further discussed in this paper.

In 2014, the well known Berkley Packet Filter (BPF) kernel facility, to filter network packets in the kernel-space, has been rewritten and extended [5]. An extended Berkley Packet Filter (eBPF) [6] [7] program is a small snippet of code that is compiled to byte code by a just-in-time (JIT) compiler. It gets loaded into the kernel, which then executes this code in a dedicated virtual machine, explicitly handling only this code. Before this code is loaded by the kernel, a pre-verifier checks the code to avoid malicious code to be executed in kernel-space - i.e. it is checked, whether the program contains out-of-bounds memory accesses, loops or global variables. Loops require to be rolled out explicitly and global variables require to be stored in memory maps, that are shared with the respective user-space application.

In 2016, a patch set for high performance networking has been added [8]. The so called eXpress data path (XDP) has been merged in the Linux kernel in version 4.8. This new approach deals with network packets taken right from the NIC, before the kernel is setting up a socket buffer structure, and rejects unwanted, passes desired or redirects packets. A good example for the power of XDP is the defense of a denial-of-service attack. When such an attack is noticed it is possible to drop the packets inside the NIC with such an eBPF program. Thus, the CPU does not have to deal with them and the system stays operational. To use this feature, however, a network driver has to support XDP programs, which can then be accessed via the newly introduced `AF_XDP` socket type - with specialized hardware, the offloading of eBPF programs to the hardware is possible. But even without driver support for XDP programs, it can make sense to use XDP in software mode, as shown in figure 1. Driver mode has been described above. The software mode uses the network driver and allocates a socket buffer structure. For the given example XDP would not make much sense. On the other hand it enables new ways of preprocessing network packets, which can save a significant amount of CPU time for other tasks.

2. CONCEPT

We investigate two different use cases: AVB Listener JACK Client (*jackd_listener*) and JACK AVB Media Clock Listener Backend (*avbmcl*). Both use cases have the same bottle neck with different consequences for the application.

The first use case is our proof of concept, since it involves all required functionalities: Integration into the build system, preprocessing of AVTP packets and sharing data between kernel- and user-space via memory maps, i.e. the audio samples. As build system the Linux native `make` is used. Both the existing application and the provided tutorials for XDP use the `make` build system [10]. The pre-processing involves three steps. In the first step, AVTP packets shall be filtered on arrival for their destination MAC address and stream ID. This step shall drop any packet that does not match the criteria and prevents a lot of context switches to the host applications waiting raw Ethernet socket. The second step is to store the audio samples contained in the AVTP packets in its integer representation to a memory map, that can be shared with the host application

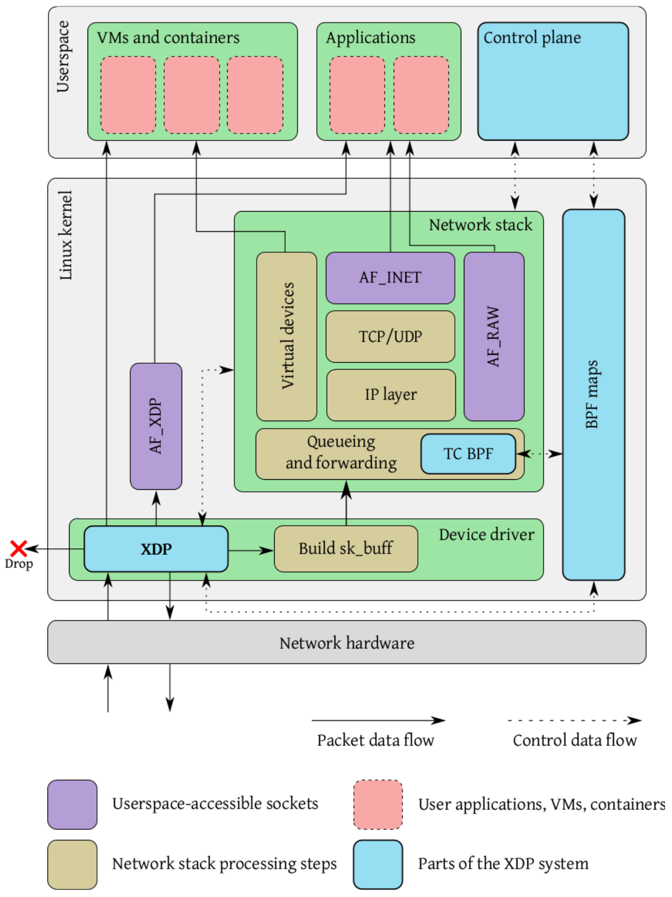

Figure 1: *Components of the XDP subsystem are shown in light blue and reside in the device driver as well as the network stack [9].*

in user-space. The final step is to pass the last AVTP packet, whose contained samples are required to completely fill a sample buffer to the host applications raw Ethernet socket. The raw Ethernet socket in the host application *jackd_listener* is waiting, it receives only AVTP packets for its own registered destination MAC address and stream ID. In fact only the last received AVTP packet is passed. On the reception of the last AVTP packet, it reads the integer-formated audio samples from the memory map, converts them to float format and writes them to a JACK ring buffer.

The second use case requires the integration of the XDP eBPF build process into the `Waf` build system [11], since `Waf` is the build system that is used to build JACK. The pre-processing involves two steps, namely the first and the second step of the first use case - filtering for destination MAC address and stream ID and passing only the last AVTP packet of a sample buffer period.

3. REALIZATION

A prerequisite for XDP and eBPF to work is a kernel later than version 4.8. We deployed a customized real-time kernel of version 5.2.17-rt9 in our test environment.

The implementation of an eBPF program with the host appli-

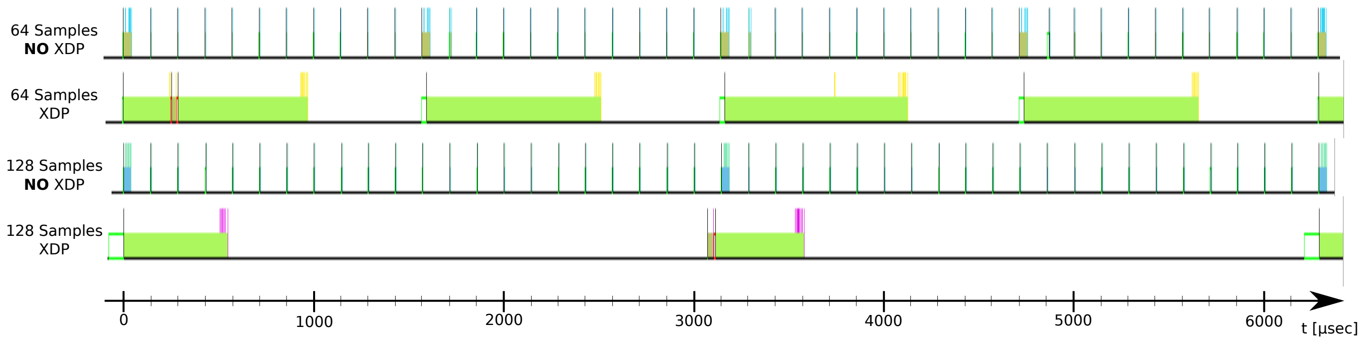

Figure 2: *Kernel traces showing the context switches of the JACK sound server for 64 and 128 samples per buffer with and without XDP.*

cation *jack_listener* has been used as proof of concept. An integration in the `make` build system already existed, which only required adoption to the host application. The `Waf` build system that is used for JACK, however, does not support the LLVM compiler framework [12]. Furthermore, it had not been possible to integrate the loading process of the eBPF program object file into the JACK backend. `Libbpf` [13] needs to find the `main` symbol of the application it is linked against, which could not be achieved until now. Thus it is necessary to compile the eBPF program in a preparing step and load it with a stand alone loader. If the `make` build system is used directly, as is the case for the *jackd_listener* application, the memory maps can be accessed by the host application via a file descriptor and a name string.

The eBPF kernel programs need to be customized, configured and compiled for each application that uses it. In which way parameters can be changed during runtime is still open for investigation.

After the eBPF program has been successfully hooked to the desired NIC, the generic command `ip link show 'dev'` can be used to verify this, i.e. the last line of the following console output.

```
$ ip link show enp5s0

enp5s0: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP>
        mtu 1500 xdpgeneric qdisc mq state UP mode
        DEFAULT group default qlen 1000
 link/ether a0:36:9f:bd:95:46 brd ff:ff:ff:ff:ff:ff
 prog/xdp id 217
```

Regarding our first described use case the current lack of financial and in turn hardware resources prevents the required implementations with corresponding complexity: Theoretically a significant number of concurrent AVB listeners may be deployed on a single server, however, in order to test it our test environment lacks a significant amount of AVB talkers. A test would comprise the resulting streams to trigger the corresponding amount of interrupts by the NIC, each of which would be pre-processed by the eBPF XDP program that has been installed for that stream's listener application. Therefore, the evaluation of a setup, in which the *jackd_listener* application benefits from XDP is not possible at the moment. In contrast our second described use case represents a solid base for the technical implementation and evaluation as described in section 4.

4. EVALUATION AND DISCUSSION

The kernel network stack needs to keep working, although a filter is implemented with XDP. This might sound obvious, but it is a development experience worth noting. It is important for the XDP program to pass all Ethernet frames up to the kernel network stack even if they are not subject to our intentions. Filtering for a specific AVTP stream for example, requires PTP and MRP to keep receiving packets, otherwise the NIC does support neither IEEE 802.1AS nor IEEE 802.1Qat. Thus, such packets need to be passed up to the kernel stack and cannot be filtered, e.g. for debugging purposes. This becomes even more important when multiple XDP programs are attached to the same NIC. It has to be ensured that those XDP programs do not interfere with each other by filtering packets the other XDP program requires for its successful operations.

The runtime optimizations provided by the XDP eBPF are realized with kernel traces. A comparison of the context-switch scheduling events of the Linux kernel task scheduler is shown in figure 2. It shows the JACK sound server process with the *avb-mcl* backend configured at 64 and 128 samples per buffer, both with and without a XDP filter program attached to the AVB NIC. The impact of the XDP programs can be seen clearly. When a XDP program is attached to the NIC, fewer context-switches take place, which provides more CPU time to the JACK clients. The JACK clients context-switches are represented by the spikes at the end of the JACK sound server context-switches at the beginning of an audio cycle. Without XDP, those spikes appear much earlier in the cycle and have less time to complete, namely until the next JACK sound server context-switch ≈ 125 μsec later.

During situations with heavy load generated by the entire system in a production scenario, the XDP improvements provide a much more robust signal processing and audio signal routing. No buffer over- or underruns occur. An in depth evaluation, however, does not provide further insights and is therefore omitted.

5. CONCLUSIONS

Integration into the `Waf` build system used for JACK is not possible at the moment, because `Waf` is not able to use the LLVM compiler framework.

The lack of ability to perform floating point operations in the kernel-space, is a limitation for further applications of XDP, i.e. for the first discussed use case. Otherwise, it would be possible to directly write the float-formated audio samples to the JACK ring buffer and eliminate any user-space interaction.

The JACK AVB audio stream listeners do not suffer from the asynchronicity between the JACK sound server and the AVB media clock, since multi threading and the JACK ring buffers decouple the two clock domains. In a scenario where a massive amount of listeners is required, listener with XDP programs in place might be an improvement to listeners without XDP. This is still open for investigation.

For the JACK AVB media clock backend, XDP provides a significant improvement and solves the context-switching problems under load. Further investigations, however, revealed that this performance could as well be achieved with an appropriate handling of a generic raw socket. Thus, XDP represents a powerful and interesting alternative butaspects such as tedious debugging, lack of floating-point operations and the retrieval of hardware timestamps outweigh the benefits significantly.

6. FUTURE WORK

The workflow to create eBPF programs has to be improved. The name for each eBPF program, that shall be loaded, has to be unique in order for the host program to correctly address the memory map for the kernel-/user-space interactions. Furthermore, the parameters required at runtime, such as the stream ID, destination MAC address and sample buffer size, need to be passed to the eBPF program at runtime. Only then can JACK change internal parameters without the need for a newly compiled eBPF program.

At the moment, it is not possible to access hardware timestamps inside an XDP program. This is on the road map of the development teams, however, it might provide further optimization for an AVB network stack in the future.

In theory, XDP would allow to use a NIC (AVB is not required for this) with a raw Ethernet socket to implement a custom protocol. This way it may be used as an interface for digital signal processors that are equipped with an Ethernet interface as well. Signal routing could be done with XDP, so that the signal processing computations are offloaded to the digital signal processor. An user-space application would only manage the audio streams. This approach will be investigated in the future.

7. REFERENCES

[1] (2019, Feb. 8) Soundjack - a realtime communication solution. [Online]. Available: http://http://www.soundjack.eu

[2] A. Carôt, U. Krämer, and G. Schuller, “Network music performance (nmp) in narrow band networks,” in *in Proceedings of the 120th AES convention, Paris, France*. Audio Engineering Society, May 20–23, 2006.

[3] (2019, Feb. 8) Jack audio connection kit. [Online]. Available: https://jackaudio.org

[4] C. Kuhr and A. Carôt, “A jack sound server backend to synchronize to an ieee 1722 avtp media clock stream,” in *Proceedings of the Linux Audio Conference 2019*. Stanford, CA USA: Linuxaudio.org, Mar. 23–26, 2019.

[5] J. Corbet. (2014, Sep. 24) Linux weekly news (lwn.net): The bpf system call api, version 14. [Online]. Available: https://lwn.net/Articles/612878/

[6] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, “Performance implications of packet filtering with linux ebpf,” in *2018 30th International Teletraffic Congress (ITC 30)*, vol. 01, Sep. 2018, pp. 209–217.

[7] (2019, Dec. 12) Bpf and xdp reference guide. [Online]. Available: https://cilium.readthedocs.io/en/latest/bpf/#bpf-and-xdp-reference-guide

[8] J. Corbet. (2016, Apr. 4) Linux weekly news (lwn.net): Early packet drop — and more — with bpf. [Online]. Available: https://lwn.net/Articles/315941/

[9] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, “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*, ser. CoNEXT ’18. New York, NY, USA: ACM, 2018, pp. 54–66. [Online]. Available: http://doi.acm.org/10.1145/3281411.3281443

[10] (2019, Dec. 12) xdp-project - xdp-tutorial. [Online]. Available: https://github.com/xdp-project/xdp-tutorial

[11] (2019, Dec. 31) Waf 2.0.18 - the meta build system. [Online]. Available: https://waf.io

[12] (2019, Dec. 31) Waf 2.0.18 documentation - waf tools - compiler_c. [Online]. Available: https://waf.io/apidocs/tools/compiler_c.html

[13] (2019, Dec. 12) libbpf. [Online]. Available: https://github.com/libbpf/libbpf