On reliable execution of applications for Linux using the approach of in-kernel processing

Student
Ba Que, Le, B.Sc
Email: b.le@tu-berlin.de

Department of Telecommunication Systems
Technical University of Berlin
To get the degree in

Bachelor of Science
Informatik (Computer Science)

Submitted

Board:

Chair:	Prof. Dr.-Ing. habil. Thomas Magedanz (TU Berlin)
Reviewer:	Prof. Dr.-Ing. Axel Küpper (TU Berlin)

January 2022

Affidavit

I hereby declare that the following thesis “On reliable execution of applications for Linux using the approach of in-kernel processing” has been written only by the undersigned and without any assistance from third parties.

Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

Berlin, 18. January 2022

ACKNOWLEDGMENTS

This research topic was offered by Next Generation Networks chair, TU Berlin. We thank our supervisor, Varun Gowtham, who provided insight and expertise that greatly assisted the research as well as guided us through all the research's phases, although he may not agree with all of the interpretations/conclusions of this paper. We thank Prof. Thomas Magedanz for comments that greatly improve the final version. We are also immensely grateful to family and friends for their support and comments at the finishing stage of the document, although any errors are our own and should not tarnish the reputations of these esteemed persons.

Berlin, 18. January 2022

ABSTRACT

Time Critical Applications (TCAs) are predictable software applications that play critical role in many areas of our world. Those kind of application rely on satisfying strict time and resource constraints, and therefore need to be carefully designed. However, recent advances in technology have promoted an interest in writing and running the TCA application on a general-purpose hardware and low-cost open source software, such that the system encompassed by TCA is flexible and could potentially replace proprietary solutions. In this thesis, the capability and limitation of running TCA on Linux - a general-purpose Operating System - using *in-kernel processing* approach will be studied and presented. In order to compare with existing approaches and verify the theory, example applications using this approach will be built and used to produce measurements such as latency and throughput. From the obtained data, the approach proved to be a promising upgrade for Linux's real-time applications that can deliver up to 10x performance improvement without permanent modification to the Operating System (OS), which we consider to be a significant advantage over the other approaches.

ZUSAMMENFASSUNG

Time Critical Applications (TCA, Zeitkritische Anwendungen) sind vorhersehbare Softwareanwendungen, die in vielen Bereichen unserer Welt eine entscheidende Rolle spielen. Solche Anwendungen sind auf die Einhaltung strenger Zeit- und Ressourcenbeschränkungen angewiesen und müssen daher sorgfältig entworfen werden. Aktuelle technologische Fortschritte haben jedoch das Interesse geweckt, die TCA für Allzweckhardware und Open-Source-Software zu schreiben und damit auszuführen, sodass das von TCA umfasste System flexibel ist und möglicherweise proprietäre Lösungen ersetzen könnte. In dieser Abschlussarbeit werden die Möglichkeiten und Einschränkungen der Ausführung von TCA unter Linux - einem universellen Betriebssystem - unter Verwendung eines In-Kernel-Processing-Ansatzes untersucht und vorgestellt. Um einen Vergleich mit bestehenden Ansätzen zu ermöglichen und die Theorie zu verifizieren, werden Beispielanwendungen mit diesem Ansatz erstellt und zur Messung von Latenz und Durchsatz verwendet. Anhand der gewonnenen Daten erwies sich der Ansatz als vielversprechendes Upgrade für Linux Real-time Applications (Echtzeitanwendungen), das eine bis zu 10-fache Leistungssteigerung ohne dauerhafte Modifikation des Betriebssystems liefern kann, was wir als erheblichen Vorteil gegenüber anderen Ansätzen betrachten.

TABLE OF CONTENTS

List of Figures

[1] https://rt.wiki.kernel.org/index.php/Main_Page

List of Listings

List of Tables

[2] https://github.com/AlDanial/cloc

CHAPTER 1

INTRODUCTION

Reliable execution of application for Linux is an important topic that is related to many infrastructure and production applications. Despite of not being designed for, Linux can be used as a platform for determinism purpose with full benefit from an open source, free and well supported operating system. In this chapter, we will introduce the issues regarding reliable execution on Linux and the state-of-the-art solutions; our approach to the problem and how to evaluate the hypothesis. At the end, the structure of this document will be presented.

1.1 Background and Motivation

1.1.1 Intro to the holistic problem

In this document, the reliable execution of applications expresses how stable or predictable the computer application's execution is, regarding execution time. In other words, if the application's execution time stays unvarying even in undesirable condition, the execution is then considered *reliable*. The execution is the action to run a computer program (or application) on a computer to solve a problem, hence the reliability comes from the whole *system* that comprises both the program and the platform it runs on. The terms *real-time system* or *time-sensitive system* are often pointed to the same definition, although they might carry minor nuances of meaning: *time-sensitive system* is a system that is sensitive to execution time variance; *reliable execution system* is a system that has execution reliability capability and a *real-time system* would require strict reliability from its program and platform. From now on, we will refer to *reliable execution* as the definition while the term *real-time system* is used to refer to systems that have a high

expectation of reliability in execution time.

Reliable execution has a close relationship to *execution time*. Execution time is the time needed for a computer application to complete its tasks. Each task usually has a start time and a *deadline*, which literally means the time the result should be produced. If the result can still be considered useful after the deadline has passed, the deadline is called *soft*, otherwise it is *firm*. If a missing firm deadline causes *severe consequences*, the deadline is called *hard*. If a Real Time System (RTS) needs to handle at least one hard deadline, the system is called *hard real-time system* [p20]. In such systems, reliability is the key to accomplish the mission. The absolute execution time (how exactly *long* the computing lasted) becomes the secondary factor since an unreliable system is unpredictable and can cause missing deadlines while an *acceptably slow but stable execution* makes **planning and designing** possible.

As such, embedded systems are commonly seen as the platform for reliable application. The systems are often compact (in term of functionality) and were purposely built with selected hardware and software: from as small as microprocessors to as immense as industrial controllers; from as simple as motor-controllers to as complex as server network switches that can handle server-graded Gbit traffic. They can be found in either professional gear or daily commodity products: electrical commodities and appliances, nonlinear compensation mechanisms, complex automation systems and adaptive control systems [p22]. Most embedded systems are simple and can be programmed directly onto the hardware, i.e microcontroller and microprocessor, Field-Programmable Gate Array (FPGA), Digital Signal Processor (DSP), Application-Specific Integrated Circuit (ASIC). More complex real-time systems might require a real-time Operating System (OS) that has guaranteed real-time capability, for instance, the open-source FreeRTOS[1] or proprietary VxWorks[2].

Generally, embedded systems are kept simple to avoid complexity that might introduce possible errors [p4] which, unfortunately, prevents building complex and/or high performance systems. These purpose-built systems usually have only limited flexibility and extendibility due to many factors, i.e designed with limited hardware resource headroom (just enough for the mission, i.e microcontroller), or use none-standard or proprietary hardware and software. Therefore, updating or repurposing purpose-built systems are generally not as fast and simple as with consumer computers or even not practical due to cost or unavailable, obsoleted components [p19], not to mention rising product cost due to development for none-standard technology and expensive hardware parts. In today's standard, many systems are required to have communication capability and thus would 1. require extra networking software/hardware component and 2. raise security concern - which is challenging for vendors to keep up with frequently updates and security fixes to be released.

During the past decades, interest in using commodity software and hardware for time-sensitive application has increased, with Linux being the most promising candidate for a real-time OS platform. Being open, free and customizable allows Linux to be the dominating OS for many application areas such as server or embedded platforms and to receive enormous support from software and hardware vendors (Figure 1.1). However,

[1] https://www.freertos.org/
[2] https://www.windriver.com/products/vxworks

Linux has struggled to find its position in the real-time industry - which has strict requirements regarding timing - due to its general-purpose-built nature. Usually, real-time systems are built to handle their targeted problems. Their priority is to maximize one or several *features or characteristics*, i.e solving the problem in the most efficient way, and/or reducing cost/size/power usage ... To reach these targets, compromises must usually be made, i.e sacrifice *flexibility and low-cost* for products that need maximum performance, or *flexibility and performance* for more efficient power consumption and lower cost. In contrast to that, Linux is a general-purpose OS designed for personal computers. This difference leads to the difference in capability and application areas: focusing on improving average system performance and user interface response instead of minimizing execution time for a few particular real-time tasks, with an average response delay for interactive processes lays between 50 and 150 milliseconds [p6]. Aiming to run on personal computers, Linux has to support a wide range of hardware specifications and provide an unified software interface for applications to run on. This leads to adding costly layers of abstraction that simplifies the complexity and reduces the knowledge required for each layer to develop and maintain, thus however greatly increasing the overall complexity of the *whole system*.

By changesets			**By lines changed**		
Intel	1454	10.2%	Realtek	97237	12.2%
(Unknown)	1196	8.4%	Intel	72565	9.1%
Google	932	6.6%	AMD	67076	8.4%
(None)	781	5.5%	Facebook	50894	6.4%
Red Hat	765	5.4%	(Unknown)	43152	5.4%
AMD	682	4.8%	(None)	40389	5.0%
Facebook	641	4.5%	Linaro	39428	4.9%
Linaro	592	4.2%	NVIDIA	38898	4.9%
NVIDIA	463	3.3%	Google	35871	4.5%
Huawei Technologies	422	3.0%	Red Hat	23312	2.9%
SUSE	311	2.2%	Marvell	19136	2.4%
Oracle	294	2.1%	MediaTek	15399	1.9%
IBM	274	1.9%	Code Aurora Forum	14564	1.8%
(Consultant)	266	1.9%	Anyfi Networks	13901	1.7%
Canonical	249	1.8%	Renesas Electronics	12888	1.6%
Arm	244	1.7%	SUSE	10940	1.4%
Baidu	234	1.6%	IBM	10808	1.4%
Renesas Electronics	221	1.6%	Huawei Technologies	10378	1.3%
MediaTek	199	1.4%	Cirrus Logic	10046	1.3%
Code Aurora Forum	192	1.4%	Oracle	8728	1.1%

Figure 1.1: Most active employers contributed to the 5.16 Linux kernel, among 251 employers that were able to identify [m16]. These giant enterprises invest together for the common benefits: hardware companies develop drivers and optimize the kernel for their products, and software companies use Linux as the platform for their main applications.

There are very complex real-time systems that can be found on important, mission critical devices on aircraft, satellite, spacecraft. Their importance, top requirements,

low-volume and/or high cost characteristics make Linux's advantages irrelevant and therefore these kind of systems are not in our scope. We refer to *purpose-built system* here as a fairly complex system that has the critical application runs on a stack of selected OS and hardware to solve a specific problem. Although this specified design helped solving the problem efficiently, these systems are usually inflexible and use non-standard hardware and software components, thus make adapting to new requirement challenging. In this case, the flexibility of Linux - a general-purpose kernel - is not a disadvantage but rather the opposite, i.e being able to adapt to new tasks and supporting a wide range of hardware and software. Recent improvement in hardware can also supplement the kernel's overheads while modern software techniques allow building complex application that can run more efficient. Last but not least, the industry is adopting many new emerged concepts and trends, which *are built or based on* Linux-based operating systems: hardware and software disaggregation; Software-defined networking (SDN); open-source-based common platform (Open vSwitch[3] Open Switch[4], ...); emigration to open source solution to avoid vendor locking. With all these conditions, now is perhaps the best time for Linux to join and secure a position in the real-time market.

1.1.2 Background

Originally, the term *Linux* refers to the *Linux kernel*, however, because the kernel alone is generally useless for end-users, this document will mention *Linux* as a **Linux based operating system** or so called a **Distro**, i.e Debian[5], Ubuntu[6], Fedora[7].

An OS is a software abstract layer that was initially responsible for managing and distributing hardware resources, although a modern OS is capable of many more features such as providing graphic user interface, multi-tasks, multi-threads and multi-users, ... In most OS architectures, the fundamental functionalities assemble the core of the OS - which is called *kernel*. The other less important features - usually in the form of *computer programs* (also called *application*) - are built around this kernel to construct a full operating system. A single application is a computer program that interacts with the OS for necessary resources to solve its assigned problems and produce result (which the user will perceive in text/audio/video format). In fact, the kernel consisted of many carefully designed applications that cooperate and communicate with each other at the very low-level and high efficiency.

These kernel's applications and user's applications run in distinguished memory *spaces*, called respectively *kernel space* and *user space*. This is because the memory representation in the OS is not how the memory is physically hard-wired, but in an *abstracted perception* that segregated the hardware memory into isolated kernel-memory and user-memory. This helps protecting the kernel program from being accessed by malicious user applications and provides better *process isolation* [p6]. Applications in these *spaces* communicate and transfer data inter- and intra-spaces through data channel and system's Application Programming Interface (API), although many are limited to only privileged users.

[3] https://docs.openvswitch.org/en/latest/intro/install/userspace/
[4] https://www.openswitch.net/
[5] https://www.debian.org/
[6] https://ubuntu.com/
[7] https://getfedora.org/

This separation design allows building complex, safe and dynamic systems with multiple concurrent users and applications without demanding tech-heavily-knowledge from the developers. Each process runs freely in its own allocated resources without effecting neighbor processes while being managed and monitored by the OS, which can perform intervention measurements such as changing priority, putting to sleep, forcing to quit, ... Limiting access to protected resources can protect the system from illegal accesses, enables concurrently sharing for common resources while still gives certain users freedom, assuming they know the responsibility and consequence [8].

Unfortunately, this design also introduces some significant overheads:

- Maintaining multiple spaces and communication between them are time- and resource-wise expensive.
- A simple task can involve several other system calls (Listing 1.1).
- User tasks have lower priority compared to system tasks: to ensure stability and functionality for a complex system, the kernel must process many important background tasks. The kernel's processes can therefore unpredictably spawn and negatively effect the user's application by preempting its process, blocking resources, ...

In any OS, access to resources is organized and planned by a component called *scheduler*. With multiple tasks existing in the system simultaneously, the *scheduler* has to decide carefully *which task can acquire the resources* and *the duration of claim* to distribute the resources evenly and maximize the designed system's target. In a real-time system, real-time tasks should have highest priority over all other tasks to accomplish the mission while in a general system, a balanced user experience is preferred and therefore might prevent a real-time task running in the system from starting and finishing in time.

Listing 1.1: Simple *chmod* command requires multiple system calls

```
1  # strace chmod 777 test.txt
2  execve("/bin/chmod", ["chmod", "777", "test.txt"], 0x7ffd62e735e0 /* 48 vars */) = 0
3  brk(NULL) = 0x5610d23c6000
4  access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
5  access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
6  openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
7  fstat(3, {st_mode=S_IFREG|0644, st_size=95672, ...}) = 0
8  mmap(NULL, 95672, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f5e6dbf9000
9  close(3) = 0
10 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
11 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
12 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20\35\2\0\0\0\0"..., 832) = 832
13 fstat(3, {st_mode=S_IFREG|0755, st_size=2030928, ...}) = 0
14 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
       x7f5e6dbf7000
15 mmap(NULL, 4131552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0
       x7f5e6d5f7000
16 mprotect(0x7f5e6d7de000, 2097152, PROT_NONE) = 0
17 mmap(0x7f5e6d9de000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
       MAP_DENYWRITE, 3, 0x1e7000) = 0x7f5e6d9de000
18 mmap(0x7f5e6d9e4000, 15072, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
       MAP_ANONYMOUS, -1, 0) = 0x7f5e6d9e4000
19 close(3) = 0
20 arch_prctl(ARCH_SET_FS, 0x7f5e6dbf8540) = 0
21 mprotect(0x7f5e6d9de000, 16384, PROT_READ) = 0
22 mprotect(0x5610d1d6d000, 4096, PROT_READ) = 0
23 mprotect(0x7f5e6dc11000, 4096, PROT_READ) = 0
```

[8] Many Unix and Unix-like OS embed these lines and show when the user runs *sudo* for the first time: 1. Respect the privacy of others. 2. Think before you type. 3. With great power comes great responsibility.

```
24  munmap(0x7f5e6dbf9000, 95672) = 0
25  brk(NULL) = 0x5610d23c6000
26  brk(0x5610d23e7000) = 0x5610d23e7000
27  openat(AT_FDCWD, "/usr/lib/locale/locale−archive", O_RDONLY|O_CLOEXEC) = 3
28  fstat(3, {st_mode=S_IFREG|0644, st_size=3004224, ...}) = 0
29  mmap(NULL, 3004224, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f5e6d319000
30  close(3) = 0
31  umask(000) = 002
32  stat("test.txt", {st_mode=S_IFREG|0777, st_size=0, ...}) = 0
33  fchmodat(AT_FDCWD, "test.txt", 0777) = 0
34  close(1) = 0
35  close(2) = 0
36  exit_group(0) = ?
37  +++ exited with 0 +++
```

The *scheduler* in a normal OS is not designed to fully prioritize real-time applications over system's tasks and is not used for real-time purpose due to their unfairness [p21]. The lack of kernel preemption (Figure 1.2) - which allows a real-time task to block certain system tasks when needed [p1] - could cause long blocking time and hurt real-time performance. Even when the real-time task is allowed to run, it might be forced stop by the other system's tasks in the next execution slot. Factors such as timer resolution, scheduling jitter and non-preemptive sections of the OS also cause unpredictable OS delay between *execution being requested* and *execution being granted* [p1]. Nonetheless, to help protect the OS from possible user's malicious code, the memory for kernel tasks is segregated from those for user tasks (called kernel and user space respectively) on modern OSs and thereby static overhead will be generated from state switching, communication and information exchanging [p24], which together contribute in a relative real-time hostile platform.

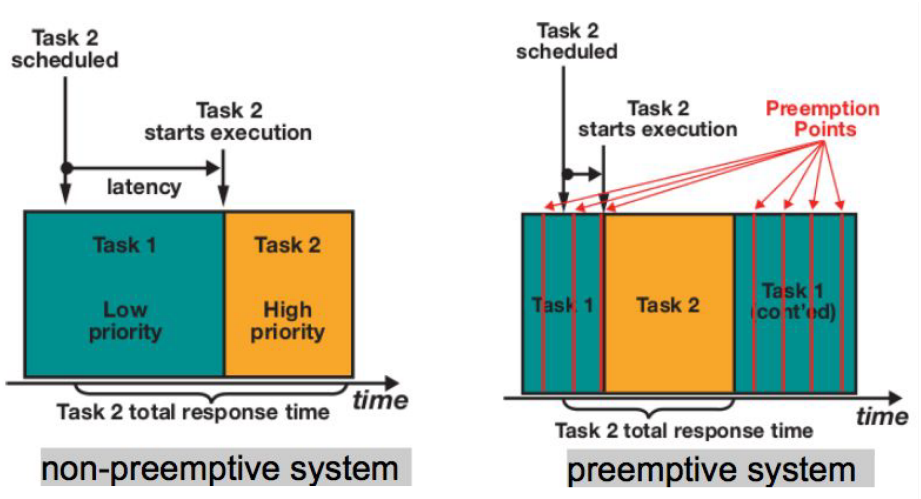


Figure 1.2: Comparison preemptive tasks [t8]. In normal systems, high priority task has to wait for current task to finish its time-slot, while that running task in a preemptive system can be stopped at any preemption-points during the time-slot execution.

1.1.3 Existing solutions

Multiple solutions have been implemented to improve real-time capability on general-purpose OS. *Dual-kernel* solutions such as *RTLinux*[9], *RTAI*[10], *Xenomai*[11] run normal OS on top of a real-time operating system core that manages real-time tasks, however heavy modifications in the kernel are required and thus changes how tasks interact with the kernel, which can make real-time tasks tricky to analyze [p26]. *Kernel patching* approach has been around since the early 2000's with the most maintained project is *Preempt RT*[12] using now by *UbuntuStudio*[13]. By applying the patch, the Linux kernel will be converted into a fully preemptive and faster responding one, includes a higher solution software clock [t7], however the patch also breaks the compatibility with mainline kernel. *Kernel bypassing*, on the other hand, totally liberates these tasks from being managed or scheduled by the kernel. Unfortunately, a huge framework is required to wrap the real-time tasks and isolate them from the host's OS, which would again bring the compatibility and maintenance into question.

With such limitations of these well-known solutions, *in-kernel processing* approach - in which real-time activities are moved into the OS's kernel space - seem to be a promising replacement. This approach moves the execution of some real-time logic in the kernel space, which allows the real-time tasks to be treated equally as ordinary kernel tasks and therefore can have the benefits of running in kernel space: earlier events processing thanks to events and handlers being kept together in kernel space; high priority; and no more communicating and exchanging between user and kernel space, which result in lower OS's delay and reduced overheads.

As a matter of fact, running user code in kernel space involves risk of compromising the system, include leaking memory and crashing [p11] [m6]. Toolings for developing and debugging kernel code are limited; convenient user-space libraries are unavailable, which might result in high cost and long developing time. Nonetheless, *in-kernel processing* is one of the most promising approach assuming the integrity and stability of the OS is taken into account. For instance, although *dual kernel* or *kernel patching* might offer top-of-the-line performance, the amount of work for maintaining patches or modifying kernel for each mainline kernel release is enormous not to mention the application itself needs to be tested. Slow paths *bypassing*, while could help reducing delay, makes many kernel's tools and services excluded, i.e debugging or logging because the real-time application section is invisible to the kernel. *In-kernel processing* on the other hand, is supported by the kernel using native APIs and methods, regardless kernel version (except major update), such as *kernel module* or *eBPF*. Most of the device drivers for Linux are *kernel modules* [t12] and *eBPF* - before being ported to Linux - is introduced and released back in 1993 for BSD OS [p23] with intensive usage since then, hence stability and safety are virtually guaranteed if implemented right. Many applications using this approach can be found in the Linux kernel and cover high performance fields such as driver or debug, tracing or filtering [m2], suggests that similar success especially in real-time application field could also be achieved. In fact, *eBPF* and some *kernel module* applications were used for filtering and load-balancer

[9] http://cs.uccs.edu/~cchow/pub/rtl/doc/html/GettingStarted/
[10] https://www.rtai.org/
[11] http://www.xenomai.org/
[12] https://rt.wiki.kernel.org/index.php/Main_Page
[13] https://help.ubuntu.com/community/UbuntuStudio/RealTimeKernel

purpose by Facebook [m5] [m15] - social network with traffic ranked 7th in the world according to Alexa ranking[14] - to power the network load balancer used in the company's infrastructure.

1.2 Target and Scope

We believe Linux systems have the potential to replace or complement existing custom hardware-software solutions for real-time purpose. By improving the kernel's real-time capability with approaches such as patching or bypassing, a Linux system can compete with a custom system in term of performance. However, these approaches might not be suitable for many development teams, because they are either in-compatible with Linux mainstream, deprecated, or involving complicated software stack. These will demand extra work, increase development time and result in higher cost and less maintained product. Instead, we promote in-kernel processing approach for modern Linux based, reliable execution systems. The approach aims to process the real-time tasks whenever possible in the kernel space, so the real-time program can take advantage of the kernel's priority and privileges. Delivering the real-time logic in the kernel can either be done using Kernel module or eBPF method that are provided by Linux kernel, with no permanent changes to the operating system. This approach eliminates the need for permanent kernel modification or third-party library and thus keeps it compatible with upstream version, which is critical for being able to receive feature and security updates.

In the up coming phases, the approach will be examined and evaluated against the requirements and the criteria. The evaluation phase is indispensable to verify the hypothesis and discover the approach's characteristics, such as capability or limitation. To accomplish this, we will develop 2 example applications using in-kernel processing methods and conduct tests with different conditions and configurations to gather data. These data will be processed and compared to the performance of Linux as well as other approaches.

Each example application consisted of the following components:

- Kernel Part: contains the in-kernel logic. We implement our interceptor and processing unit here.
- User-space part: this is where the rest of the application are implemented. In our application, this part will receive data from the kernel part for further processing.
- Communication: includes transmitting data and messages between the components.
- Log: we have logging mechanism embedded in each of our program parts to keep records during the run-time of the application.

Although our vision for the application remains unchanged, the requirements for the end-product applications have to be adjusted during the course of development due to the variance in task's priority, unforeseen circumstances or modifications in technical details. Figure 1.4 illustrates an ideal but minimal-features application in production.

[14] https://www.alexa.com/topsites

1.3 Structure of the Thesis

In Chapter 2, the related work and information to comprehend the topic will be presented. This includes the existing approaches and several Linux's technical details needed to understand the problems that they addressed. The requirements for the evaluation process will be analyzed in Chapter 3. We divide these requirements based on their purposes: **Functional requirements** are used to assess *if the approach can improve the performance over Linux*, and **Business requirements** decide *if it is worth considering whether the approach would be suitable for production, based on factors such as cost or legal*. In Chapter 4 we will introduce the example applications, their components and the test cases. This will cover the reasons for the final design decisions of the example applications, and how they will be used for testing and producing data in Chapter 8. Chapter 5 will report the preparation for the development which involves setting up environment and creating helper programs/tools. Chapter 6 and Chapter 7 walk through the implementation details of Kernel-Module- and eBPF-based applications, respectively. Each chapter will provide more technical details related to the current used method (Kernel Module, or eBPF), and produces both the blueprints and the actual code for that example application from the general design discussed in Chapter 4. Afterwards, the evaluation process will be presented in Chapter 8. We will show the used setup for running the test cases, each has different conditions and configurations. The applications developed in Chapter 6 and Chapter 7 will be run along with the normal application (created in Chapter 5) on the testbed, and the gathered data during the test phase is used to evaluate against the requirements in Chapter 3. Finally, we will conclude the research in Chapter 9 with the summary and further work sections.

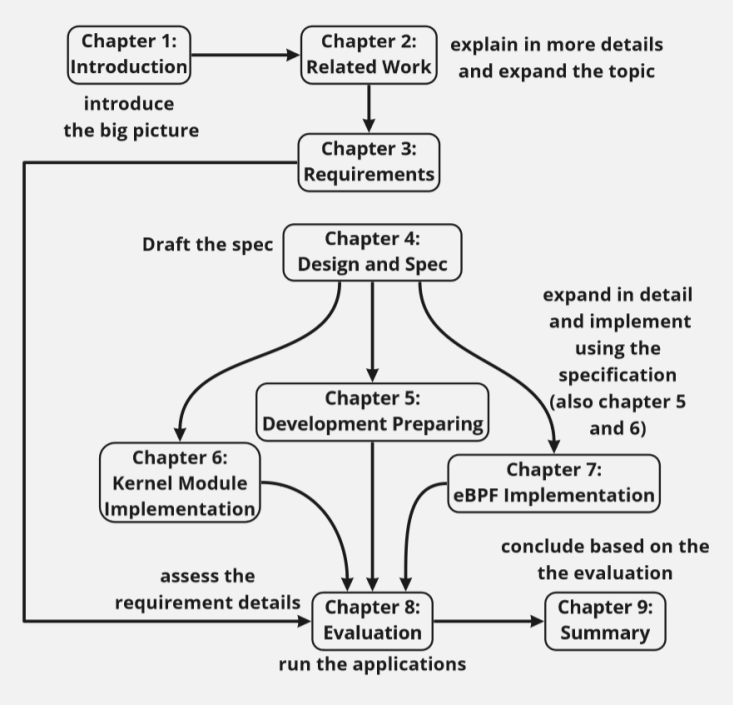


Figure 1.3: How the chapters relate to the others.

Figure 1.4: The approach with its ideal components. The future work is listed in the **Optimizations** and **Todos** leafs. Only the features defined in Section 1.2 can be implemented and thus leaves the other listed in the leaf **Components** open

CHAPTER 2

RELATED WORK

Having different design philosophies, purpose-built OS is much more suitable for reliable applications, however a general OS like Linux would have advantages when it comes to cost, support and available resources due to its popularity. Using a general OS for reliable purpose demands changes to be made to improve predictability and determinism. In this section, we will provide insight into the problem's root and the most well-known intervention approaches for Linux to increase its real-time capability, including our *in-kernel processing* approach and the reason why it was chosen over the others.

2.1 Background and Motivation

A computer system consisted of hardware and software with the Central Processing Unit (CPU) acts as the brain, which is responsible for processing the instructions provided by the programs running in the system. Although a program can be written to directly communicate with the hardware, a software abstract layer is typically placed in the middle: a simple micro-controller is equipped with a thin run-time to abstract away the low level communication, while a personal computer would need a complete OS that can manage hardware devices and provide resources or services for multiple higher-level

applications.

There are multiple software applications running simultaneously in a computer: system applications that are parts of the OS itself, and applications that belong to the users. Despite of the multi-tasking illusion, the CPU (core) can only process one task after another, yet it can process very fast in a short spin (or also called *quantum duration*, usually ranges between 10-100 miliseconds [p2]) and switch to other task in the *task queue* that had been prepared by the OS's *scheduler*. This dynamic list tells the CPU which task shall get processed next and is updated frequently by the scheduler using different strategies (Linux calls them *Scheduling policies*, i.e *SCHED_FIFO* for **first-in-first-out**, *SCHED_RR* for **Round-robin** [t25]). The scheduler makes decision based on current policy and factors such as task's *deadline*, *arrival time*, *priority*, ...

Back in the early days of computer programing, the programers are familiar with the details of the hardware and therefore can optimize their programs to make the best out of the platform. Later on, increasing in complexity of the software product pushes the developers to be more concerned with problem-solving and let the OS handle the architecture details [p10]. This *outsourcing* is achieved by providing many abstractions at the OS level: certain hardware details are hidden and resources are provided in generic term on request, i.e memory is managed in virtual pages while processes see it as a continuos segment [t28]. Most OSs split the memory into distinguished spaces, namely *kernel space* and *user space* as presented in Figure 2.1. System applications and other low-level code (such as driver) run in the kernel space while user's applications live in the user space. User-space processes run under restriction: access to normal resources is granted through system calls and privileged actions must be done in kernel space on behalf of user's task [p5]. Although this 2-spaces design has some negative effects on Linux's real-time capability, we need to understand the architecture and idea that shaped the kernel, and how they effect reliable execution. In Section 2.1.1 we will briefly introduce the kernel's design and philosophy. Section 2.1.2 discusses the consequences on reliability of time-critical application. Section 2.2 presents the existing solutions as well as their pros and cons, which are the reason why we come up with the in-kernel processing approach introduced in Section 2.4.

2.1.1 Kernel design motivation and philosophy

Linux is an open source, general purpose kernel started by Linus Torvalds back in the 90s. As a common knowledge, a general purpose OS serves as a platform for other applications and provides an interface to the user (graphically or terminal-based). In fact, the operating system is a collection of software and toolings wrapped around a *core* - or *kernel*, which performs the most fundamental tasks, such as Process scheduling, Memory management, Provision of a file system, Creation and termination of process, Access to devices, Networking and Providing system call (or APIs) [p18] ...

Primitive computers had their memory hard-wired with the processor in a known configuration (bus width, size). The OS therefore manages memory using the hardware address, and all running applications would share the same address space. In a modern OS, memory assigned to a process has *virtual address space*, which doesn't reflect the actual address of the memory in hardware and will then be translated into the actual address by the Memory Management Unit (MMU) in the processor. The usage of distinguished virtual address and memory space allows having *machine independence*,

program modularity (by supporting programs consisted of unlinked-modules) and *list processing* (supporting structured data) capability [p10]. Other benefits can be named [p10] [p6]:

- Supporting larger applications: large program can still be run by only loading partial executable code of the program into memory, page-wise.
- Multiple programs can be maintained in the cache, because process only loads pages of programs instead of full executable blob.
- Sharing library between processes.
- Kernel can increase the number of maintaining processes in the system.

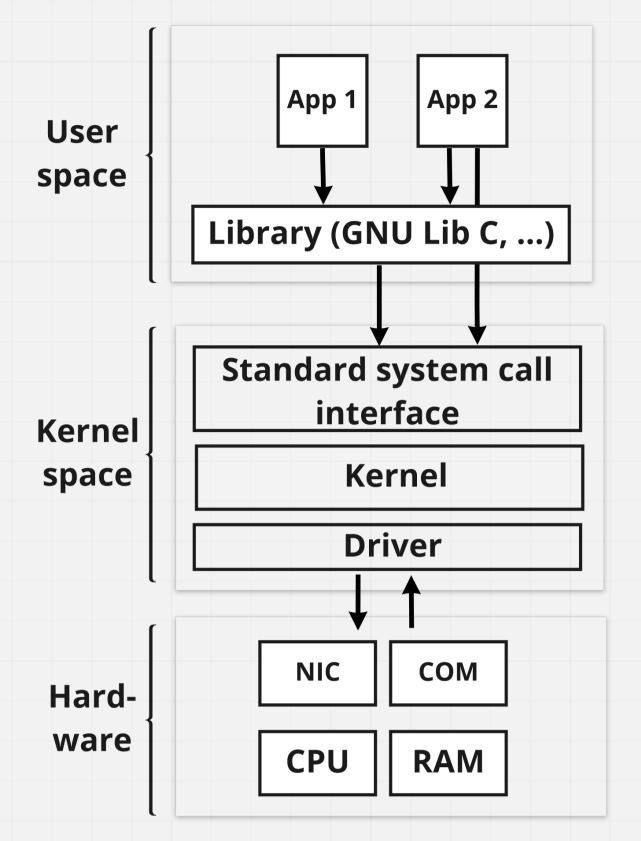


Figure 2.1: Structure of a typical Linux OS. The applications running in the user-space request access to resources by calling system calls through the standard library, and the kernel communicates with the hardware through the driver.

Privilege is also a feature that is supported by any modern OS. First added in the Intel's 80286 processor [p15] in 1982, *Protected Mode* (also *Supervisor Mode*) is the hardware state of the processor where it is allowed to access certain protected memory areas. There are different levels of privilege on the x86 architecture: user processes

(unprivileged) run in Ring 3 while system processes have the most privilege and run in Ring 0.

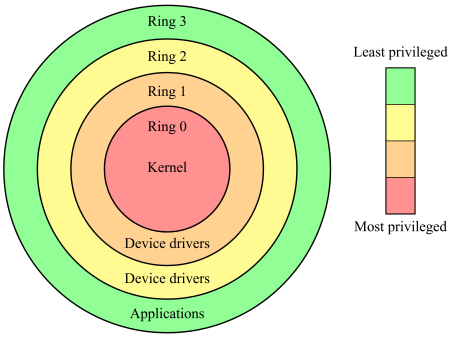


Figure 2.2: Privilege rings for the x86 [m4]

In Linux, these privilege modes are called *kernel mode* (where processes are *trusted*) and *user mode*, where only un-privilege memory is accessible. With hardware support (typically through the CPU's integrated MMU), the hardware memory is virtualized (hence the term *virtual memory*, which distinct a memory unit's *physical location* from its *virtualized address* in the system) and segregated in *kernel space* and *user space* for reasons such as security (due to isolation). The reserved memory for the system is called *low memory* while the user memory is called *high memory* [p16].

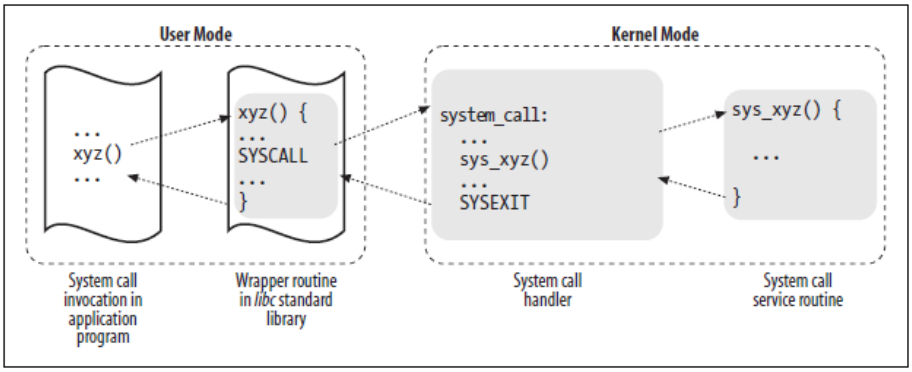


Figure 2.3: Invoking a system call from user-space [p6].

During the application's running time, there are needs to involve certain privileged tasks, i.e accessing devices, querying system information ... The user space application calls the matched *system call* - also *API*[1] - to dispatch the request to that privileged task. If the application has accordant permission (for example, run by sudo or root user if the system call is protected), the kernel will signal the CPU to move the program into the kernel space and execute the needed instructions on behalf of the user (Figure 2.3).

[1] "... The former (API) is a function definition that specifies how to obtain a given service, while the latter (systemcall) is an explicit request to the kernel made via a software interrupt ... From the programmer's point of view, the distinction between an API and a system call is irrelevant ..." - Bovet & Cassetti. (2000). *Understanding the Linux Kernel* [p6]

When the system call returns, the processing mode will be switched back to user mode and the application can continue executing. During the switching, the application is blocked until the system call has returned. This creates significant overheads and delays due to complex chain of calling and the expensive switching mode, waiting for result, paging between protected and unprotected memory [p6].

A process is an independent processing-entity belongs to a user, which can be understood as "an instance of a computer program that is currently being executed" [t27]. Each process is separated from the others, has its own virtual address space and can have permission to access certain resources [t19]. To distribute system's resources between running processes, Linux kernel has a *scheduler* that is responsible for allocating and assigning access to resources. The latest version of scheduler on Linux is named CFS , stands for *Completely Fair Scheduler* and is designed for desktop environment that models an ideal, precise multi-tasking CPU [t5]. It ensures a good overall user-experience, however doesn't guarantee a constant amount of resource for any process [t23] and thus makes determinism on Linux impracticable.

2.1.2 Time-critical application with Linux

Time critical application - or also called real-time application - is an application that takes reliability execution very seriously. Unlike the understanding *real-time* as *high speed and fast responsive* known by the public, the reliability (regarding processing time) is in-effect the most important characteristic for real-time systems. Such system expects the result to be returned in-time, and thus, there will be no planning available without the determinate timing capability. Real-time systems usually run on purpose-built hardware for maximum performance (i.e embedded controller in breaking system, airbag or collision avoidance), or on systems with hardened real-time OS (i.e FreeRTOS or proprietary VxWorks), however they all have their own limitation and cost as discussed in Chapter 1.

A general purpose OS is designed to balance between responsiveness and performance for general use cases, with Windows, Linux, macOS OS as the most common OSs for Personal Computers (PC). Great demand, affordable price and ease of use allowed general-purpose OS and Personal Computer (PC) to be dominant in the consumer hardware and software market. In the last few years, the interest in using Commercial off-the-shelf (COTS) systems in replacement of specific-built systems for time-sensitive applications has increased. This is due to the fact that software and hardware have become more capable of handling such tasks in respect of performance, maturity and support. With Software Defined Application (SDA) becoming a new trend, COTS solutions have a great chance to secure a position in the time-sensitive domain, because unlike custom solution, once COTS software and hardware have reached the point where the difference in performance become insignificant, their availability, affordability along with flexibility would be more suitable for a fully connecting and moving world of SDAs. Complex real-time applications will benefit the most from the OS's libraries and ecosystem with a wide range of supported software and hardware - especially the support for multi-cores and processors - which is a great opportunity to significantly reduce the product's cost, development effort and time-to-market [p26].

Even so, there are obstacles that are preventing general-purpose OS from being widely adopted for time-sensitive applications, mainly the insufficient predictability of the general-purpose OS for real-time tasks caused by the kernel [p26]. Normally,

general-purpose OS is not explicitly aware if a running task in the user space has "real-time" requirements, nor has accordant mechanics to ensure their timely completion, namely fully-preemptive kernel, better locking mechanisms, suitable interrupt handlers or high-resolution clock [t7]. Without these mechanisms, processes on a general-purpose OS may suffer unpredictable delay spikes [p13] [p21] (especially when the system is under load as shown in Figure 2.4), thus makes the platform inappropriate for real-time purposes. And despite of latency improvement (thanks to the advance in hardware computation power and enhancement in software design) from the order of milliseconds in the 2000s to microseconds in 2019 [p26], the additional and inevitable support for features such as complex processor (multi-level cache, hyperthreading, ...), virtualization and countermeasure for new security threats has made general-purpose OSs challenging again for real-time support [p7] [p25].

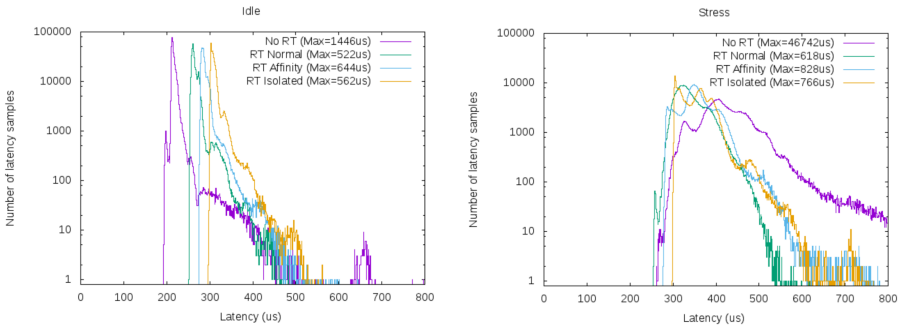


Figure 2.4: Comparison of real-time ethernet round-trip-time histograms for system idling and under load between vanilla kernel and patched kernel [p13]. The violet lines represent the number of latency values tested on system with normal Linux kernel. Notice that this line indicates significant more high-peak values in the stress test (right graph) compared to the other real-time modified systems.

Improving real-time capability of general-purpose OS can be done in many ways. Perhaps the most radical approach is running a modified kernel on a real-time OS - called *dual kernel* approach with examples are *RTAI* or *Xenomai* project. This way, the real-time tasks will be handled by the hard real-time OS while normal tasks run on the general-purpose OS. Less progressive, the kernel can be modified in such way that it behaves similar to a hard real-time OS but not too different from the vanilla kernel. This approach - called *kernel patching* - is widely adopted due to its simplistic and good performance [t3] even under load (Figure 2.4), although the patches might cause update and long-term-support issues[2]. On the other hand, *kernel bypassing* approach solves the problem by not touching the kernel but taking over the related processing logic from the kernel-space and moving them in the user-space for processing in a pre-allocated, controlled environment, thus removes unnecessary kernel's overheads. Each of these approach has its own pros and cons, which will be investigated further in the following Section 2.2.

[2] "The development of a particular version usually stops when the focus switches to the next mainline version. This happens once a new stable candidate is released." - Linux Foundation DokuWiki [t22]

2.2 Existing Approaches

In the above section we have reviewed the motivation, problems and briefly introduced some solutions. In this section, more details about these solutions will be explained along with their advantages and disadvantages. The *in-kernel processing* approach will be presented later in the Section 2.4.

2.2.1 Dual Kernel

Dual-kernel eliminates kernel latency by running real-time and normal tasks on different run-times. In this kind of system, Linux kernel runs on a thin real-time OS, which directly handles interrupts and manages the hardware. Interrupts for normal tasks will be forwarded to Linux(where they might be delayed by the kernel), whereas real-time tasks can directly receive the interrupts. Depending on the implementation, the real-time tasks can run in the Linux user space as in *RTAI* or in a POSIX emulator as in *Xenomai*. This method has been used or experimented for critical application with impressive results, i.e executing an interrupt handler in under 35 microseconds on RTLinux running on a generic x86 - which was stated to be hardware limit in the 2000s - compared to 600-20.000 milliseconds on standard Linux [p29].

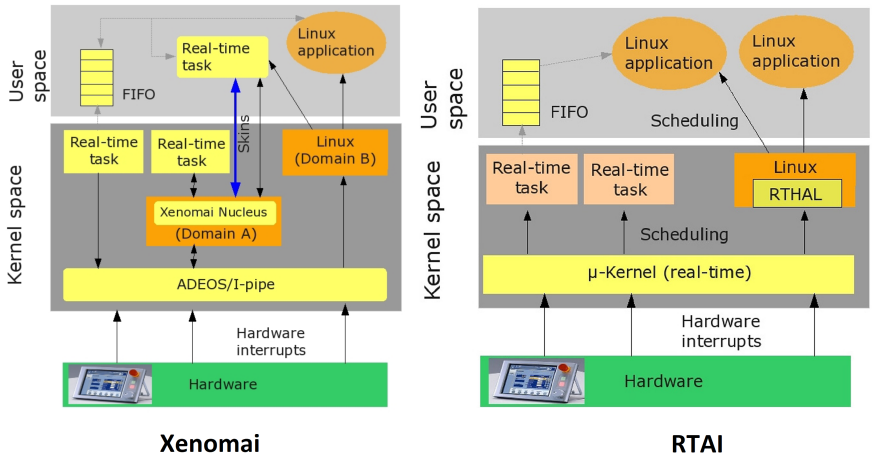
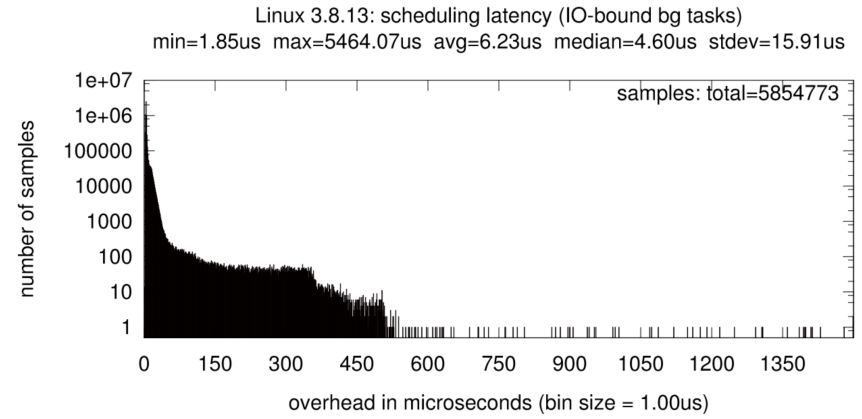


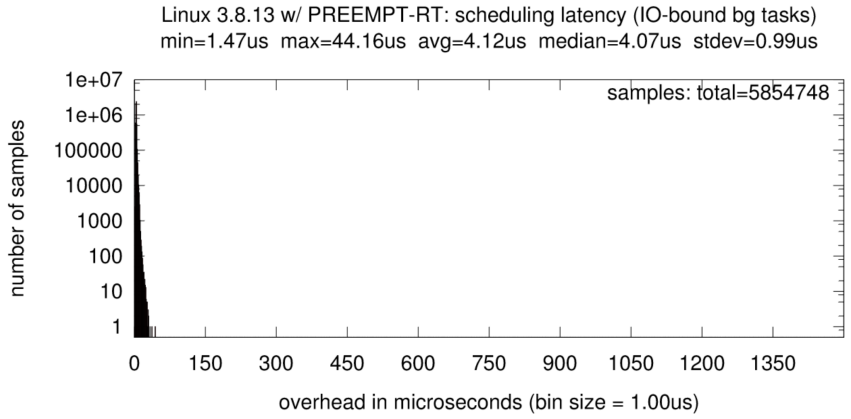
Figure 2.5: Structure of Xenomai and RTAI dual-kernel [p3]

Because of the run time where real-time tasks run on, real-time applications may not be able to use the Linux kernel along with other problems, such as special tools and libraries required for the developing and debugging processes. Running a hard real-time OS coexists with a modified kernel also raises compatibility issues [p3] [t24] and makes real-time tasks tricky to analyze [p26].

2.2.2 Kernel patching



Linux 3.8.13 without the PREEMPT_RT patch



Linux 3.8.13 with the PREEMPT_RT patch

Figure 2.6: Scheduling latency in the presence of an I/O-bound background workload on 3.8 vanilla Linux and a patched system with *Preempt RT* [p9]. On the patched system, most of the scheduling latency values concentrate in the 0-40us range compared to 0-500us on the normal system.

In this case, the kernel code will be modified with a patch (or a set of patches) to fix several positions that effect real-time performance. For instance, *Preempt RT* - the most known and adapted patch - works by replacing *spinlocks* with *rtmutexes*, converting interrupt handlers into preemptive kernel threads, turning several critical sections to preemptive and providing high resolution timers in user space [t7]. Although the performance was significantly improved (Figure 2.6) and even on par with *dual-kernel* method in many aspects [p27], *kernel patching* comes with its own drawbacks, namely throughput degradation [p27] or breaking compatibility with the mainline kernel. Ap-

plying patches also requires recompiling and restarting the system, which can negatively effect uptime or stability.

2.2.3 Kernel bypassing

Not intended to reduce the overheads caused by the kernel, the methods of *kernel bypassing* approach avoid the problems by isolating real-time tasks from the system. This approach found itself very useful in networking area where the Linux's network stack offers poor performance for many real-time applications, either low through put (1 million packets per core [m10] compared to several millions with DPDK [t26]) and high, unpredictable latency (from a few microseconds to several milliseconds under load, depending on the test condition and kernel version) [p12] [p8] [t2].

One of the projects that followed this approach is PF_RING published in 2014, introduced a new type of socket called *PF-RING socket*. This socket allows user-space applications to speak directly with the PF-RING kernel module, which in turn intercepts and stores packets in a ring buffer [p8]. Other projects can be named such as *Netmap*[3] with kernel modules that can deeply integrate with hardware, or *Snabbswitch*[4] that segregates network card from the kernel and implements its own network stack.

Perhaps the most well known bypassing method is the DPDK framework, developed by Intel and has been taken over by the Linux Foundation. Similar to other methods, DPDK makes use of pre-allocated hardware resources (include the NIC), direct memory access (DMA) and uses other techniques and practices to ensure best performance of its class such as cache alignment, core affinity, disabling interrupts, huge pages and many others [p8]. DPDK comes with rich built-in libraries and manages NICs by itself using its own driver set, which however limits supported vendors and models and completely detaches the devices from the kernel. The framework runs in user space and requires no specific configuration, supports a wide range of hardware from common vendors and can scale well on multi-core or multi NUMA nodes systems, with reported to be able to saturate 50Gbit line [t26].

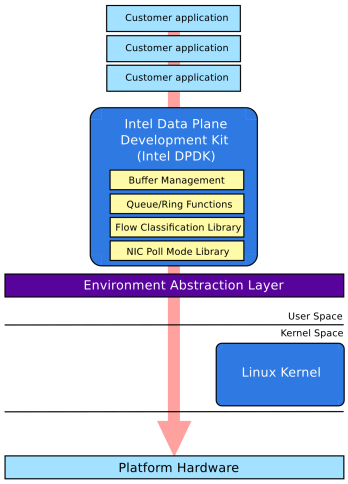


Figure 2.7: Major DPDK components [p8]

Nonetheless, this approach also faces several challenges. For networking purpose, replacing the well-known, battle-tested kernel network stack with a custom one might cause more problems and require extra unnecessary work [p14]. Isolating could also

[3] http://info.iet.unipi.it/~luigi/netmap/
[4] https://github.com/snabbco/snabb

restrict the application from accessing useful helpers of the platform. Extensive knowledge of the system is required for developing a sufficiently isolated environment (these introduced methods are huge projects with many low-level interventions) which can greatly reduce the number of development teams that can afford a custom solution. The available projects are mostly designed for network-related application, where avoiding system's default software stack is feasible and can significant improve the real-time performance. So far, *kernel bypassing* has no other notable real-time method that we are aware of, except the *PCIe passthrough* method. This method allows hardware to be assigned (*pass through host OS*) directly to the guest OS to avoid virtualization overheads [5][6][7]. The passthrough method is common seen in server field and could possibly find its application in real-time field for niche market, like using real-time hardware in a virtualized environment.

2.3 Problem Gap

Linux is a popular OS for both personal and professional users with abundant support from hardware and software vendors. It also powers some of our most important infrastructures: internet servers, super computers, workstations, ... Because of its success, one might suggest using Linux as the OS for running time-critical application such as a real-time system. However, the design decision that allowed Linux to reach its current popularity now prevents it from being utilized in time-critical system: general-purpose, optimize for general workload with limited support for preemptive.

As mentioned in the previous sections, there existed a handful of approaches and methods to improve real-time capability of Linux that can deliver comparable performance over purpose-built solution, although they are rarely used in real-world applications due to:

- Invasive modification to kernel and results in upstream-incompatibility. This prevents performing system's minor update and major upgrade, which leaves the system in an out-of-date and vulnerable state.
- Introducing complex framework.

Execution of applications in the user-space can't guarantee determinism due to non-constant accessing to requested resources. This would raise the questions: if reliability cannot be achieved in user-space, how would processing these tasks in the kernel space perform? Is there any viable option to achieve such model? We will answer these questions in the next section where the in-kernel processing approach and its methods are introduced.

2.4 Introducing In-kernel processing Approach

Unknown to the user, the CPU keeps processing background tasks at any given moment, even when the system has entered in idle state. Depending on the tasks, the

[5] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/virtualization/chap-virtualization-pci_passthrough
[6] https://wiki.archlinux.org/title/PCI_passthrough_via_OVMF
[7] https://pve.proxmox.com/wiki/PCI(e)_Passthrough

amount of time the CPU spends in the kernel- and user-space might vary as showed in Listing 2.1. Regardless user-space activities, the background tasks - or kernel tasks - will be prioritized over, as showed in the combined load. Therefore, moving tasks into kernel-space will be most meaningful to the process that involves inter-space communication, low-level events or stable execution (which will disfavor interruption caused by scheduler).

Listing 2.1: CPU time with different loads showed by *top*. Amount of time the CPU spent in kernel space and user space is marked with ***sy*** and ***us*** respectively

```
1  # stress --cpu 4
2  # This stress command will run CPU-bound processes in user-space
3  ...
4  %Cpu(s): 99.9 us, 0.1 sy
5  ...
6
7  # yes test
8  # This command repeatedly prints the "test" line on the standard output.
9  # IO activities require lots of kernel time.
10 ...
11 %Cpu(s): 11.1 us, 13.4 sy
12 ...
13
14 ### Combined of the 2 commands.
15 # Note that the kernel time is roughly the same as if only "yes" command was called.
16 ...
17 %Cpu(s): 73.1 us, 14.9 sy
18 ...
```

For the system stability and safety, kernel task and user task live in different spaces, and intentionally injecting custom code into the kernel space might as well put the system at risk. Extending the kernel can be done either by changing the source code, using kernel module, or using eBPF. Modifying the source code doesn't benefit us: same as kernel patching approach, it breaks the compatibility and requires opening modified source code if the binary file is externally distributed due to the GPL2 license [t15], which many industry partners might feel uncomfortable with. On the other hand, Kernel module and eBPF are the perfect methods that allow custom code to be loaded into the kernel space on demand, are supported by the kernel, and are used to develop performance applications: low-level intervention, driver, debugging, ... Although both methods demand the custom code to be licensed under GPL conditions, we will discuss about this problem and how we overcome this in the evaluation phase (Chapter 8).

We are interested in *in-kernel processing* model for Linux real-time application because of its ease to deploy, compatibility and performance. These could be very helpful for applications such as IoT, where mass deployment, maintenance and keeping the system up-to-date are extremely important due to the development cost and security concerning. Supporting for a great number of devices and diverse types might be challenging even for the manufacturer with limitless resources, along with their always-connect nature makes them the perfect targets for hackers. Windows and macOS are also general-purpose OS, although they are specifically designed for Desktop users and lack the necessary customization for systems of various sizes. These OSs require expensive licenses, lots of resources to run and therefore are not suitable for our vision of real-time system.

2.4.1 Kernel module

Traditionally, to inject code into the kernel space, a *kernel module* will be inserted into the kernel and can be unloaded later. Many OSs support *kernel module* under different names, such as *kernel module* in Linux, *kernel loadable module* in FreeBSD, *kernel extension (kext)* in macOS (now deprecated), *kernel extension module* in AIX, *kernel-mode driver* in Windows NT and *downloadable kernel module* (DKM) in VxWorks Kernel Module[8]. *Kernel module* provides the ability to extend the kernel without re-compiling and rebooting [t10] [t29], however it could threat the stability of the system with bad implementation, which could easily result in total system crash [m1]. Kernel module is usually used to extend the kernel's feature set, such as create Linux drivers and other kernel's components.

2.4.2 eBPF

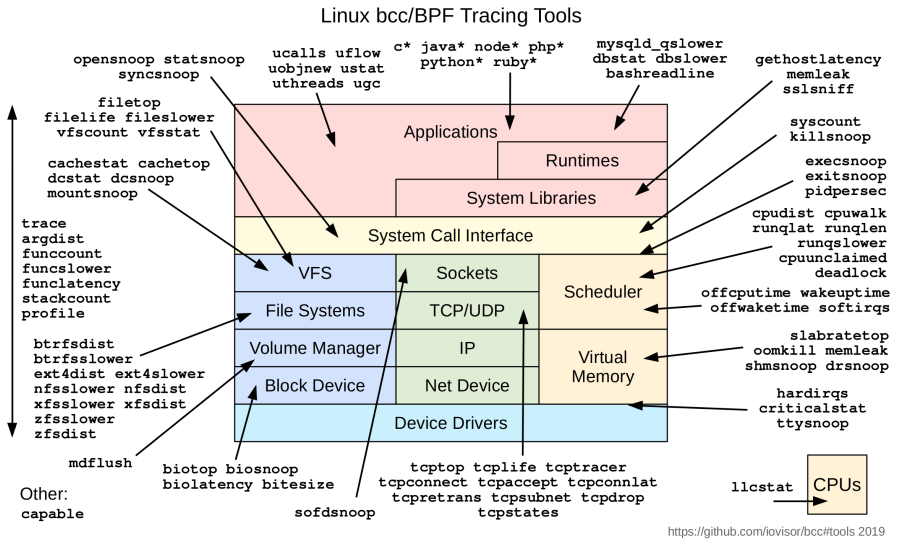


Figure 2.8: A wide range of Linux eBPF-based tracing tools [t1].

On Linux and BSD-family OS, there are other methods can be used to insert custom code into the kernel, namely extended BPF (eBPF) and BSD Packet Filter (BPF). On BSD, BPF is a technology which was introduced 2 decades ago while eBPF is the extended, re-implemented version for Linux, developed in the 2010s. Instead of running custom code directly as in *kernel module*, BPF and eBPF will carefully inspect and run the code in a virtual machine to prevent any program with defects from crashing the system. This method has been quickly adopted in the kernel development community with many new tools introduced and existing tools reworked, with few of them are showed in Figure 2.8. Nowadays, eBPF technology is being used in production by big tech: firewall, filter, load-balancer in Facebook [m15]; monitoring and analyzing in Netflix [m9]; security and observation in Cilium Project [t6]; cloud infrastructure in Alibaba [m8].

[8] https://en.wikipedia.org/wiki/Loadable_kernel_module

An application using eBPF often consisted of 2 programs: the kernel-program and the user-program. The kernel-program is written in a subset of C language and will be compiled to RISC assembly code that the virtual machine can understand. Language features and size of the kernel-program are limited (i.e no loops allowed), however the kernel provides several APIs and system-calls to perform fairly complex function or communicating/exchanging data with user-program. User-program is usually responsible for controlling, monitoring and also processing data transmitted by the kernel program.

CHAPTER 3

REQUIREMENT ANALYSIS

3.1 Overview

As discussed in the related section, the industry can't fully utilize mainline Linux because of poor real-time performance. Existing modifications for Linux improves this, but also harms the compatibility. To solve the problem, we introduced using Linux OS platform for reliable application with in-kernel process approach in previous chapters as an alternative that promises comparable performance with other approaches, albeit without negatively altering the kernel.

In-kernel processing approach allows certain tasks to be processed in the kernel space and thus eliminates unnecessary intermediate overheads. This approach can be used to implement real-time application for Linux operating system to improve real-time performance over Linux's default. Besides that, industry partners can benefit from the unchanged Linux kernel and its ecosystem for rich support, feature and security updates, and an existing developer community that are already familiar with Linux.

The rest of this research will verify the hypothesis by demonstrating the effectiveness and determining the characteristics of the approach. This requires building example applications to use as the test-beds for demonstrating and for data collecting. These

data will be then used to calculate the improvement over Linux's performance and to compare with other approaches, as well as to compare between the methods used in the example applications. This chapter will define the requirements for that evaluation process. We split the requirement entries into 2 categories:

- **Functional requirements**: use to assess the approach's viability and raw performance.
- **Business requirements**: requirements that are related to the decision making process, i.e comparison with competition approaches.

The requirements for each category are listed in the following table:

Table 3.1: Requirements in categories

Research target	**Requirement detail**	Code
Functional requirements: Determining approach ability	Assess performance improvement	R1A
	Assess application's compatibility	R1B
	Viable in real-world application?	R1C
Business requirements: Comparing performance	Comparison to Linux performance	R2A
	Compare to other approaches performance	R2B
	License	R2C
	Technical Support	R2D

3.2 Functional Requirements

3.2.1 Improve performance (R1A)

The approach should bring improvement over mainline kernel's performance. Unpredictable latency causes or derives from the kernel should be improved in one or many aspects:

- **Lower average latency than vanilla kernel** (reacting to events) OR **Lower processing time than vanilla kernel** (through more CPU time, less de-scheduled, less system calls ...)
- **Lower jitter, more predictable than vanilla kernel**: distribution of processing time values should be close to average values.

3.2.2 Minimal modification, improve compatibility (R1B)

No system is considered safe without being patched and updated during its lifetime. Modifying the kernel makes it hard to apply changes from mainline upstream to clients. Fortunately, the approach makes use of *kernel module* and *eBPF* to impermanently deliver the custom code, which are provided by the kernel itself and therefore *compatible* by default. The custom code might need to be updated to new specification (in most cases: effortlessly) due to occasionally breaking compatibility of kernel's internal API.

3.2.3 Approach's viability (R1C)

An idea works in the lab doesn't guarantee its success in real-world. To estimate the appliance of the approach, the research should be able to determine possible appliance fields and approach's advantages in such cases. Other factors such as development experience, available documentation or professional/commercial support can also greatly influence team's productive and product's quality. Lastly, the technology's longevity is also of great concern to product's long-term support and life-time: enterprise tends to stick with proved technology that won't fade away anytime soon.

3.3 Business Requirements

Comparison of in-kernel processing approach and other competition approaches falls into this category. In the technology world, there are usually multiple solutions existed for a particular problem with different compromises (i.e cost or complexity). The evaluation result will aid making business decision based on the advantages of the approach, performance difference and other characteristics mentioned in Section 3.2.3.

3.3.1 Comparison to Linux performance (R2A)

The performance improvement compared to Linux's default solution must be presented along with other factors: system load, memory usages, ... Along with the information provided in the *Improve performance* requirement (Section 3.2.1, R1A), we can learn about how the approach can enhance Linux, and with which cost.

3.3.2 Compare to other approaches performance (R2B)

Similarly, the research should provide a comparison to other approaches's performance. This requirement can be used to decide the most suitable method based on several factors, such as performance, resource consumption and cost incurred.

3.3.3 License (R2C)

Although we encourage using and contributing to open-source software, keeping the source code closed is understandable in certain circumstances, especially when the use-cases are highly security related, or the secrete business logics are mandatory for staying ahead of the competitors. This must be later investigated and evaluated in the Chapter 8.

3.3.4 Technical support (R2D)

As mentioned in Section 3.2.3 (Approach's viability, R1C), longevity and (professional, accountable) support often play deciding role in making technical decision. This will also be evaluated in the Chapter 8.

CHAPTER 4

DESIGNING AND SPECIFICATION FOR THE EVALUATION

This section is dedicated to introducing the architecture of the example applications, how they work and how they will be tested. Besides that, helper programs and scripts needed for the experiment will also be mentioned.

4.1 General Design

4.1.1 Idea and motivation

In-kernel processing is an approach to build computer program that requires reliability execution in replacement of custom-built software and hardware. By default, Linux kernel provides 2 code-delivery methods to inject custom code into kernel space, namely *Kernel Module* and *eBPF*. For demonstration and testing purpose, we will build 2 example applications using kernel module and eBPF method, and run those applications in different configurations and conditions to 1. Demonstrate the usage of the approach in a real-world application and 2. Take measurements for requirements evaluation and comparison.

General architecture of an in-kernel processing application:

Figure 4.1 shows the general architecture of an In-kernel processing application that we have chosen compared to a normal application. The **Normal Application** lives in the user space completely and relies on the kernel interface for requesting resources. A in-kernel application has 4 components: **User-space Part**, **Kernel Part**, **Communication** and **Log**. The **Kernel Part** can react more quickly to kernel'events and access kernel resources freely, or forward the data to the **User-space Part**. **Communication** component allows such data or message transmitting, and **Log** component is responsible for keeping records. Section 4.2 will explain more about the tasks and details of each component, with their implementation being presented in Chapter 6 and Chapter 7.

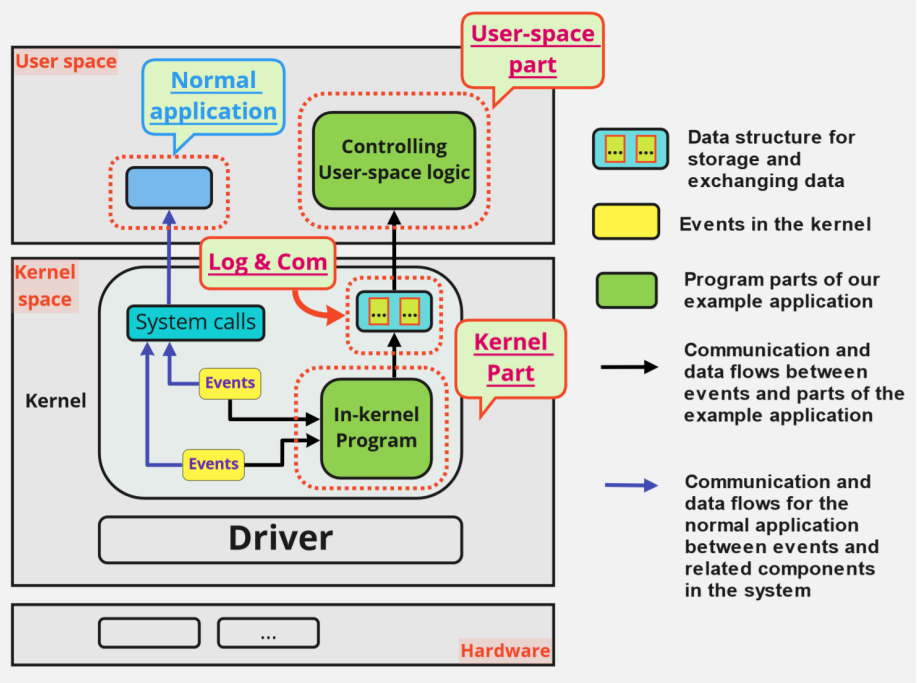


Figure 4.1: General application architecture of in-kernel processing approach application with Communication, Log, User-space and Kernel Part components.

Alternatives to the chosen architecture:

- *The application lives in the kernel space and has no user-space part*: In our vision, the user shall be able to interact with the application. Kernel program doesn't have any method to intuitively and safely communicate with the user, while an user-space program has many options, such as providing a desktop interface or hosting a web based control panel. Our hybrid design offers the best of both worlds: good performance from the kernel, and legacy of the user-space environment. The application can chose how to handle the events: processing in the kernel space, forward to the user space in fast-path using the communication component, or let the user-space part receive them as a normal application.
- *The kernel part forwards all events and data to user-space part*: An example for this is DPDK or PF_RING. Their user-space parts receive the events and process

them with allocates user-space resources. This means transmitting time is wasted if the events can be handled in the kernel space, where they were captured. In our application, a handler for this very purpose can be loaded into the kernel in a safe and dynamic manner.

Chose the topic for the example applications:

Before looking for a candidate application field and designing such application, we firstly define the application's properties:

- is sensitive to spike in execution time. The effect to the Quality of service (QoS) should be easy to observe or detect.
- has available in-user-space processing implementation. This means there are available resources to help reducing development time for the normal, user-space application such as example code, libraries, system calls.
- has real-world application. The demonstration should be as close to a real use-case as possible.

We pick network application field inspired by these conditions. The nature *latency sensitivity* of network application fits well to *reliable execution*, since having low latency and stability play deciding role in ensuring QoS for many network applications [p12] [p13] [t9]. To detect violations in stability, one can take multiple measurements and install triggers in an application that react to any abnormal latency value. In reality, many network applications require precise, periodical timing among with low latency value, such as low-level real-time applications that can be found in controller, network switch/router, ... or high-level such as streaming services. Note that the referred term *latency* is used to point to the *amount of time* to deliver data from the system's networking hardware to the application, i.e a video data packet travels from the hardware Network Interface Card (NIC) to the web browser. The travel time on-the-line is rather irrelevant.

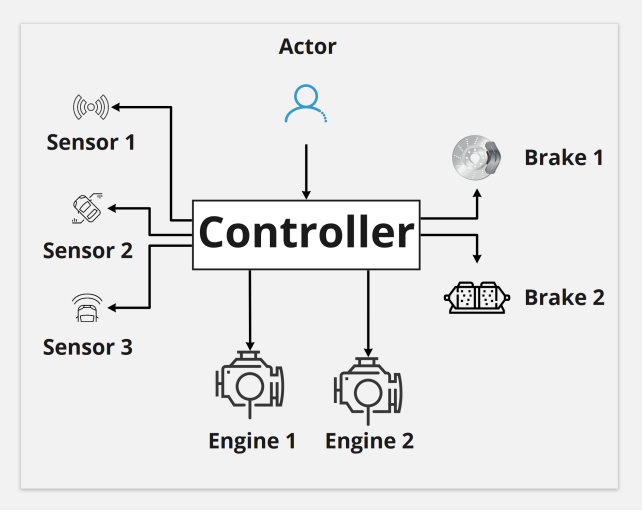


Figure 4.2: An example of the controller in its surrounding environment

We create a test scenario where example application acts as the controller that receives data packets sent by multiple clients. Clients are computers that are either attached to a component or equipped with a sensor and periodically transmits data back to the controller. The controller is a computer box running Linux OS that runs the example application and is connected to the same networks as the clients. Besides the real-time tasks, the controller is also simultaneously responsible for other none-real-time tasks such as processing data, communicating with other controllers or interacting with user. Each client sends data of types *LOG*, *DATA*, *CTRL* (controlling packets), ... with different latency-tolerant levels. Based on the data received from the clients, the controller will decide how the system should react and send the instructions to the corresponding receivers. Failure to react in-time might lead to mission failure. Therefore, the designing target for the system is delivering and processing the most important packets in time, which implies stable latency values and consistent processing ability.

Translate to implementation idea:

In a normal implementation, an application would use standard Linux socket and leave the Linux network software stack managing the delivery. However, even though a normal Linux socket is a function-rich, well-tested software stack, there are spikes in latency values that exceed the typically latency value of 10-30 microseconds (which is sustainable for most use-cases) and can go up to thousands of microseconds [t2] [p12]. The situation gets even worse if the system is busy processing more system background tasks that, unfortunately might have higher priority than our user-space, real-time application.

Instead of using standard Linux network stack, the delivering of important packets will be handled by the example applications themselves. We will implement 2 applications using 2 methods *kernel module* and *eBPF* to **intercept incoming network packets in the default packet flow of the controller**. The approach is not to be confused with the networking *bypassing approach*. The kernel-bypassing approach makes use of the same techniques as we will use, however, we embed some of the logics in the kernel space for earliest processing, and let the rest of the packets follow the bypassing flow, or the normal Linux's flow. This way, resource allocation might have less influence to the real-time business logic and waiting time can be reduced thanks to earlier access to the packet's content in the kernel space and less back-and-fort data traveling in the long path. To accomplish these features, each application must be separated into of 2 parts as showed in Figure 4.3:

- Kernel part: this part lives in the kernel space and intercepts the network packets as soon as possible, typically deep down in the Linux network stack. Here the packet can be examined and transferred to the other part. Also, various actions can be taken from this level, i.e *DROP* unwanted packets or *PASS* irrelevant packets.
- User space part: this part is a normal Linux program runs in user space. The program reads packets captured by the kernel part and also handles other tasks such as logging, communicating, reporting and interacting with users.

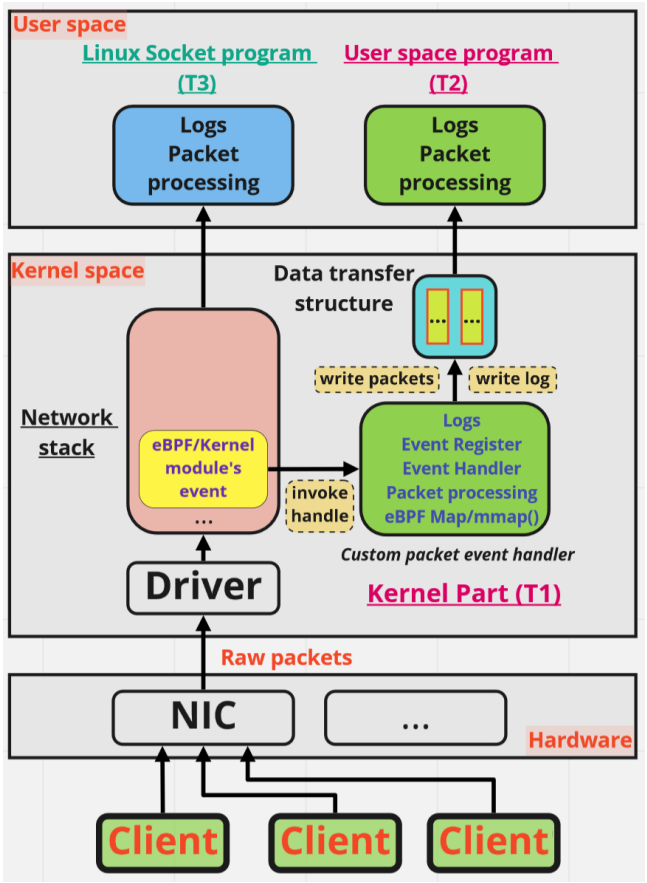


Figure 4.3: An example application and the Linux socket server running simultaneously in a test. The example application is aware of the event and the corresponding data much earlier, thanks to the in-kernel interception and process program (the custom packet handler). **Note that we only implement the ingress flow**, means the receiving end of the controller box.

4.1.2 How the tests will be conducted

A *client* (written in C) will be used to send high rate packets to port *UDP 8080* on our server to simulate multiple clients that constantly send data to *server*. Each packet is assigned with a serial ID number (a *64 bit unsigned long*) so that the server can detect packet loses by checking the missing number in the sequence. There are 3 server applications need to be implemented: 1 normal Linux socket server, 1 application implemented kernel module method and 1 application implemented eBPF method. Kernel module- and eBPF application both have kernel part and user part. The kernel part is capable of reading the packets deep inside the network stack, much earlier then Linux socket program. They also can move the packets to the user part directly, instead of using slower and longer network stack path.

A full test-suit comprised of multiple runs for each of the 2 following combinations:

- Client sends packets. Kernel module application and Linux socket server capture packets.
- Client sends packets. eBPF application and Linux socket server capture packets.

Each application has multiple measurement points that are capable of taking time measurement in nanosecond resolution. The measure points are listed as follow, as showed in Figure 4.3:

- In kernel part, where we have made sure that the packet is the type we want (T1).
- In user space part, where we have successfully fetched the packet from kernel part (T2).
- In Linux socket program, where we have read the packet from network stack (T3).

From the obtained data, we can calculate:

- How our applications perform compare to Linux socket in "best case" or "fastest case" scenario T1 versus T3, where "packets can be processed when they first enter early stage in network stack".
- In normal case: T2 versus T3. This means we compare the arrival time of Linux socket program and our user space program. We expect greatly latency reduction due to the fact that we skip the complexity of network stack.
- How eBPF and Kernel Module competes with each other.

4.2 Components in the Example Applications

Regardless method used, each example application is made of the following components and parts: kernel program, user-space programs, communication and log component. The below sections introduce shortly these components. Chapter 6 and Chapter 7 will explain the implementation of the kernel-module and eBPF based applications in details, respectively.

4.2.1 Kernel Part

Contains several logics to filter and process interested packets. The packets can be processed here or forwarded for further processing in the user-space part. The eBPF kernel program is relative limited in what system calls are available to it and which actions can be taken, while developing the kernel module program is very similar to Linux kernel development. The component has to rely on the communication component for data and messages transmission.

4.2.2 User-space Part

Contains the rest of processing logics. Other necessary functionalities are also implemented here, i.e managing and controlling the user program and the kernel program. An application in production might as well need to write data into a database and handle a web-based/desktop-based user-interface. These tasks are less real-time demanding and can be implemented as normal in user-space.

4.2.3 Communication

Kernel part and user-space part need to communicate and exchange data. Like many Linux's drivers, the kernel-module based application can use *mmap* system call to share a memory area with the user-space program. *mmap* is a Linux's system call to create a mapping in the virtual address space that other processes can access [t18]. Our kernel module program stores packets in a buffer and waits for user-space program. A */proc/intercept_mmap* file will be created by the kernel module as the access point for any user-space program. The user-space part will open and read this file using standard *read* and *write* system calls. For eBPF's data storage and exchanging purposes, Linux kernel offers *BPF map* as a generic data structure for data storage, and sharing with other eBPF kernel programs and user-space programs [t17].

In a sophisticated application for production, each program part has to be able to communicate with the others using message exchanging and signaling mechanisms, however we will not implement this feature in our example applications due to complexity and time limitation. Useful usages can be named: controlling the kernel-part from the user-space controlling interface (start, stop, reload, update new rules), or collecting statistic.

4.2.4 Log

Logs are pieces of information that software applications keep during run-time, usually for reasons such as accounting or debugging (when encounter problems). The level of log mode defines the granulate and detail of each log entry. For instance, if the user runs the application in production mode, we would only interest in the amount of requests it receives while the developer runs the same application in debug mode to collect data such as memory usage, triggered events ...

Each of our application's part collects timestamps for events in high-resolution for testing and debugging purpose. Such events are: packet arrival in kernel space and user space program, packet arrival in Linux socket program. In a normal application, log data is continuously written into temporary memory and gets flushed out (written) to disk when the buffer is close to its capacity. Due to the high-performance test scenarios, logging and exporting might degrade the performance of the application. During the tests, the kernel and user-space parts must handle tens of thousand packets per second, which could effect the application's performance. The log buffers can be quickly filled up, which would trigger an expensive write-to-disk operation and reduces the program's performance further more. Implementing a good log subsystem that can keep up with high-performance applications would require too much work, so we chose to allocate enough memory for pre-defined test and export these log entries when the application terminates. This reduces the flexibility of the program and limits the test's duration, however greatly simplifies the development.

4.3 Measurement and System Load Simulation

Testing the example applications requires methods for collecting data and simulating conditions. This section will introduce how the measurement can be done code-wisely, and how the system-load is simulated.

4.3.1 Measuring time

Clock resolution refers to the smallest amount of time that a clock can increase in each tik, or how *fine* time can be measured using this clock. On Linux, functions such as *gettimeofday*[1] (deprecated) or *clock_gettime*[2] can deliver timestamp with micro- to nano-seconds precision, depending on the implementation and the system. Because a computer is a closed, isolated system, measuring time between events is not possible with an external clock device unless aided with customization in electronic and OS level (i.e debug headers on micro-controller board). Time measurement is therefore performed using system's software clock, however delivering high-resolution, precise result is tricky duo to the competition between measurement task and other normal tasks for CPU time. Calculating average value from multiple executions and measurements in a long duration would help reducing the margin of error. These data will be recorded in log buffer, and then flushed to file for further processing. Production system can use log frameworks (such as log4c, zlog. ...) and more modern solutions (such as *perf Linux profiler*) to collect advance and low level metrics.

4.3.2 Stress test

To discover limitation or help finding unexpected behaviors, the target system needs to be pushed to limit while running the real-time task, usually using a *stress test* application which generates synthetic load. Depending on the goals, the test will simulate complete or partial system load by running IO-, network- or CPU-bound tasks. Multiple *stress test* tools and frameworks have been developed and could be relevant to our use-case, such as *stress-ng*[3], *UnixBench* for general system test; DPDK, *pktgen*[4] for network packet generating; ... We will also try to use standard test frameworks that were used in other researches for comparison purpose, such as measurement programs described in [p21].

[1] http://linux.die.net/man/2/gettimeofday
[2] https://linux.die.net/man/3/clock_gettime
[3] https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html
[4] https://www.kernel.org/doc/html/latest/networking/pktgen.html

CHAPTER 5

DEVELOPMENT PREPARATION

Previous chapters have introduced the problems and causes, related details as well as our proposal for resolving the issues. For demonstrating and testing purposes, Chapter 4 proposed the designs for the example applications. In this chapter, the details and procedure for setting development environment, running and setting the example applications - and the helper programs - will be discussed.

5.1 Development and Test Platform

Developing Kernel module is laboriously hard and slow, with even a small error can render the system in unstable state. For eBPF the working condition is much better, however the debug information is still very limited and sometimes misleading. This leads to the decision to develop and test the applications on Virtual Machines, because VM can be easily restarted in case of a system crash or not responding. If we have a major system configuration modification, a snapshot can be made and restored when needed. We use *scp* or *rsync* to copy code from dev machine to the test machine. Both machines are practically identical, so compiling can be done on either machine.

We use 2 VMs for the developing and testing the application, installed with the same Ubuntu Desktop version: Ubuntu 20.04.2 LTS 5.8.0-63-generic, which was a stable release. Kernel version can be locked as follow:

```
1  # /etc/default/grub
2  GRUB_DEFAULT="Advanced␣options␣for␣Ubuntu>Ubuntu,␣with␣Linux␣5.8.0-63-generic"
3  # sudo update-grub
```

For convenience, eBPF application will be compiled on the dev machine and binary programs will be transferred to the test machine using a *bash* script:

Listing 5.1: Compile eBPF application on dev machine and move binary to test machine

```
# Run this script on test machine.
# Compile code on dev machine
ssh -i /home/que/.ssh/id_rsa que@192.168.1.12 "cd ~/Desktop/xdp-tutorial/eBPF_measure && make
    clean && make" &&
# Copy binary programs to the test machine
rsync -avIL -e "ssh -i /home/que/.ssh/id_rsa" que@192.168.1.12:~/Desktop/xdp-tutorial/eBPF_measure/* /
    home/que/Desktop/mymodule/eBPF_measure &&
echo "Running ebpf_measure_user ..."
# Execute
sudo ./ebpf_measure_user -d ens33 --filename ebpf_measure_kern.o
```

The kernel-module program doesn't require special environment. We prepare a *Makefile* to specify compilation rules for kernel programs as well as Client and Linux socket server. *Rsync* is also used to move binary programs to the test machine. Having a *Makefile* speeds up the process by automating, although mixing *space* and *tab* caused us many problems at first.

Listing 5.2: Makefile for kernel application and helper programs

```
obj-m += kernel_module/intercept-module.o
ccflags-y := \
  -DDEBUG \
  -ggdb3 \
  -std=gnu99 \
  -Werror \
  -Wframe-larger-than=1000000000 \
  -Wno-declaration-after-statement \
  $(CCFLAGS)
# Kernel module
km:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
# Clean kernel module output
kmc:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
# User processing program for Kernel module
kmup:
	gcc -Wall kernel_module/user-processing.c -o kernel_module/user
# Clean
kmupc:
	rm kernel_module/user

# Make rules for client
c:
	gcc -Wall helper/client.c -o helper/c
cc:
	rm helper/c
# Make server
s:
	gcc -Wall helper/server.c -o helper/s
sc:
	rm helper/s
```

5.2 Acknowledgement of used Resources for the Development Purpose

In the concept testing phase, the following foreign resources were used:

- Register *rx_handler* code for kernel module by Github user G5unit[1].

[1] https://github.com/G5unit/lkm_rx_handler

- Expose shared memory from kernel module to user space by Github user cirosantilli[2].
- Initiation code and workflow from XDP-project[3].

These resources greatly helped us to cope with the steep learning curve, served as the guidelines and working examples for our own implementation and logics. Although these code can be learnt from the official source code and documentation, they undeniably sped up our development process and even take an important place in our code base.

5.3 Helper Programs and Scripts

This section introduces the programs and scripts used in all the developing, testing and evaluating phases: a simple Linux socket client as a packet sender and multiple python scripts to visualize the data.

5.3.1 Client

The client is first introduced in the design section. In real-life use-cases, client can be a computer or sensor that belongs to a machine or acts as a standalone system. These clients send data periodically to the server and might expect instruction returns.

In our test, we prepare a simple Linux socket client that sends hardcoded data to server applications. The implementation can send packets with various rates: 50.000, 6.000 and 3.000 packets per second with packet size range from 1024 bytes to 64 bytes. An unique ID number is embedded in each packet for controlling. For more serious tests, a client implemented using other testing frameworks should be considered for higher rate, i.e DPDK can easily achieve line rate of 40Gbit/s using normal Linux server [t26].

Listing 5.3: Simple client implementation

```
1  struct Payload {
2      unsigned long client_uid;
3      unsigned long uid;
4      unsigned long type;
5      unsigned long created_time; //nsec
6      unsigned long ks_time_arrival_1; //nsec
7      unsigned long ks_time_arrival_2; //nsec
8      unsigned long us_time_arrival_1; //nsec
9      unsigned long us_time_arrival_2; //nsec
10     char data[960]; // 1024B packet
11 };
12
13 struct Payload pl;
14
15 unsigned long uid = 0;
16 while(1) {
17     // unsigned long now = get_nsecs();
18     pl.uid = uid;
19     pl.created_time = get_nsecs();
20     int sent = sendto(sockfd, &pl, sizeof(struct Payload),
21         MSG_CONFIRM, (const struct sockaddr *) &servaddr, sizeof(servaddr));
22
23     uid += 1;
24 }
```

[2] https://github.com/cirosantilli/linux-kernel-module-cheat/blob/master/kernel_modules/mmap.c

[3] https://github.com/xdp-project/xdp-tutorial/tree/master/advanced03-AF_XDP

5.3.2 Linux Socket Server

A normal Linux socket server is used to produce baseline result. Similar to kernel module and eBPF user-space programs, this server also records timestamp when the packet is successfully captured and exports these data to log files when the program terminates.

Listing 5.4: Simple Linux socket server implementation

```
1  while(keep_running) {
2      socklen_t len;
3      bzero(buffer, MAXLINE);
4      len = sizeof(cliaddr);
5
6      recvfrom(sockfd, (char *)buffer, MAXLINE,
7                  MSG_WAITALL, ( struct sockaddr *) &cliaddr,
8                  &len);
9      unsigned long now = get_nsecs();
10     struct Payload pl;
11     memcpy(&pl, buffer, sizeof(struct Payload));
12     if (pl.uid <MAX_LOG_ENTRY && pl.uid >= 0) {
13         unsigned long buff_index = pl.uid / MAX_ENTRIES_PER_LOG_BUFF;
14         log_buffs[buff_index][pl.uid % MAX_ENTRIES_PER_LOG_BUFF] = now;
15     }
16 }
```

5.3.3 Script for Log Processing

We prepare 2 python script for processing log files that were produced by the applications. The scripts have to scan the log directory, identify test cases and process the content. To export graphics, we use the Python library *matplotlib*[4]. There are also many scripts written for other purposes during the implementing process whose results will not be mentioned in the evaluation phase, for instance: check for missing packets, extract certain type of log data, calculate mean and median values.

[4] https://matplotlib.org/

CHAPTER 6

KERNEL MODULE APPLICATION IMPLEMENTATION

In this chapter, the implementation details for kernel module method will be presented. First section is dedicated to introducing Linux kernel module followed up by detail explanations for kernel and user-space parts.

6.1 Introduction

6.1.1 Overview

Linux is shipped as a whole package consisted of many pre-compiled binary files. Any changes must be done in source code, compiled and loaded into the system, which would in turn create a larger kernel with longer compilation time (for Linux kernel 5.8: around 2 hours on our machine) and require a system reboot to load new changes [p28]. Fortunately, Linux kernel can use *loadable kernel module* to extend its functionalities by loading loadable binaries. The kernel modules are (pre-compiled) "pieces of code that can be loaded and unloaded into the kernel upon demand, ... (that can) extend the functionality of the kernel without the need to reboot the system" [p28]. The stock Linux

kernel already supports a huge number of real and virtual devices by loading driver modules on demand. These loaded modules can be listed using the utility program *lsmod*, which turns data from */proc/modules* into a more human-readable format). On our development system there are over 500.000 kernel module files available[1], albeit only around 60 of them are loaded after system started and thus helps save a significant amount of memory.

Table 6.1: There are 10 of 14.6 million lines of code in the Linux 5.8 source code are from the *driver* directory, or 68% lines of the source code including headers, reported by *cloc*[2] package.

Language	**files**	**blank**	**comment**	**code**
Total kernel C	28.963	2.871.350	2.351.931	14.629.004
Total kernel C/C++ Header	20.959	581.566	998.683	4.454.088
Driver C	17.088	1.982.272	1.508.379	9.998.551
Driver C/C++ Header	8.250	313.590	530.622	3.021.842

.

The file extension of a compiled kernel-module program is *.ko*. Originally, the module object file was linked to the kernel in user-space. From kernel version 2.6, the kernel took over the process and required additional information in the kernel module file. This difference is identified by the *k* character in the suffix [t11]. A kernel module program can be inserted by hand using the command *insmod test.ko* and removed with *rmmod test*. Although being linked to the kernel, this module is not a permanent part of the kernel and will not be loaded on starting by default. The module will live in the kernel space (developer calls this *dynamically linked against the kernel*) and share the same address space as other components of the kernel, which means it has access to kernel's exported *symbols*. A symbol is either a function (or in this case, system call/API) or a variable that is exported and available in the namespace. Running *cat /proc/kallsyms | wc -l* on our machine returns **145794**, which is the number of symbols exported by our kernel [p28]. These access rights grant kernel module the power, but this also comes with great responsibility: if the module crashes (i.e segfaults), the entire system might also crash. In the user space, memory leaking is often detected and fixed by the OS while kernel developers have to handle this by themselves. There is also no privilege protection since the module has become *part of the kernel*, which also means more risks being introduced.

Any distributed kernel module shall be provided under the terms of the GNU General Public License version 2, which means binary distributor is required to provide the source code [t15] on request. Although there is company that wants to make an exception (i.e Nvidia graphic driver that has GPL-bridge links to proprietary binary and is the source of many outrages and troubles in the kernel developer community [m11] [m3]), no kernel module should be held close-source unless the binary data is for internal usage only. There is also the possibility to license the program under proprietary license, however the binary will be excluded from accessing GPL-only kernel's symbols[3], which basically renders the program useless.

[1] found with *find /lib/modules/$(uname -r) -type f -name '*.ko*' -print0 | xargs -0 cat | wc -l*

[3] "The module is under a proprietary license. This string is solely for proprietary third party modules and cannot be used for modules which have their source code in the kernel tree. Modules tagged that way are tainting the kernel with the 'P' flag when loaded and the kernel module loader refuses to link such modules against symbols which are exported with EXPORT_SYMBOL_GPL()." [t16]

6.1.2 Technical details

In the view of the fact that the kernel module is an extreme complex topic, this section will focus only on the details that are the most relating to our application, such as memory management and deployment.

Linux's memory management:

As introduced, the memory in a modern OS like Linux is almost always referred by the virtual address, which will get translated into physical one using the CPU's MMU and mapping table data that the kernel maintains internally. The Linux kernel uses multiple types of addresses for different purposes, such as *User virtual addresses* (32 or 64 bits in length, depending on the underlying architecture) that are seen by user-space programs, *Physical addresses in 32 or 64 bit quantities* for communicating between the processor and memory, *Kernel logical addresses*, *Kernel virtual addresses* and *Bus addresses* [p16].

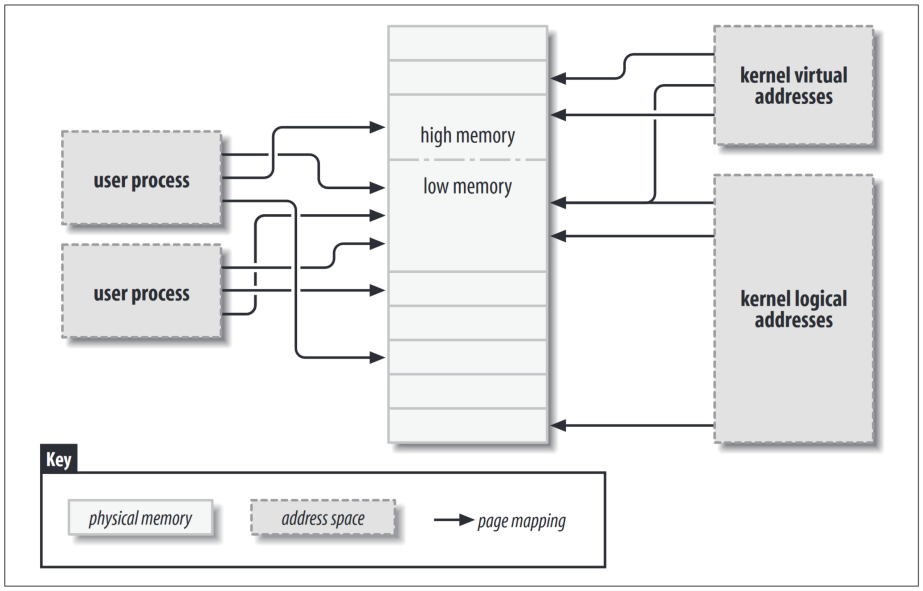


Figure 6.1: Address types used in Linux [p16]

As seen on Figure 6.1, each address points to a physical address. However, Linux mostly handles the memory on the per-page basis with each page has the size of 4KB on x86 systems. Each virtual address can be shifted *PAGE_SHIFT* bits to the right to reveal the *page frame number* [p16]. These numbers are defined in *<asm/page.h>* and can be gathered using system call, for instance: the command *getconf*[4] *PAGESIZE* would read the system variable and return **4096** on our x86 developing machine.

In kernel space, memory can be allocated by simply calling *kmalloc* (similarly to user-space counter part) or picking from a long list of APIs and flags for the allocation. The GFP acronym in the flags stands for *get free pages* and defines the

[4] https://linux.die.net/man/1/getconf

behavior of the allocator and the allocated memory [t20]: *GFP_USER* for user-space, none-moveable memory; *GFP_NOWAIT* prevents reclaiming under memory pressure; ... For even bigger memory needs, special APIs can allocate memory in page-order basis like *__get_free_pages(gfp_mask, order)* which returns a buffer containing 2^*order* pages.

Kernel module development and deployment:

Developing kernel module is similar to writing other program in the C programing language, except that the <*linux/kernel.h*> must always be included.

Listing 6.1: The *Makefile* to compile the kernel module program

```
1  obj-m += kernel_module/intercept-module.o
2
3  ccflags-y := \
4    -DDEBUG \
5    -ggdb3 \
6    -std=gnu99 \
7    -Werror \
8    -Wframe-larger-than=1000000000 \
9    -Wno-declaration-after-statement \
10   $(CCFLAGS)
11
12 km:
13         make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

Kbuild is the underneath build system for compiling kernel module. The **-C** flag followed by the path where the kernel is located and **-M** signals *kbuild* to build an external module program, followed by the path of the module [t4]. The used build-process is *out-of-tree building* and only requires kernel headers, which mostly locates in */lib/modules/$(shell uname -r)/build*. Compiling the kernel module program with this *Makefile* produces the object data *intercept-module.ko* that will then be loaded into the kernel using the command *insmod intercept-module.ko*, and unloaded using *rmmod intercept-module*. The command *modinfo intercept-module.ko* can be used to examine the kernel module file.

Debugging is indispensable during and after the development. Although *gdb* would be far more superior for debugging purpose, our example application is simple enough that most encountered problems can usually be addressed by printing messages with *printk()* - the equivalent variant of *printf()* - and analyzing the text. This function specifies a log level to write the message to, which is a kernel ring buffer that can be read in the user-space using the command *dmesg -wH* [t21].

6.1.3 Architecture of our KM Application

As stated, the kernel-module method's application consisted of 2 programs: kernel program and user-space program. Figure 6.2 shows the application's architecture, communication connections and components of the application:

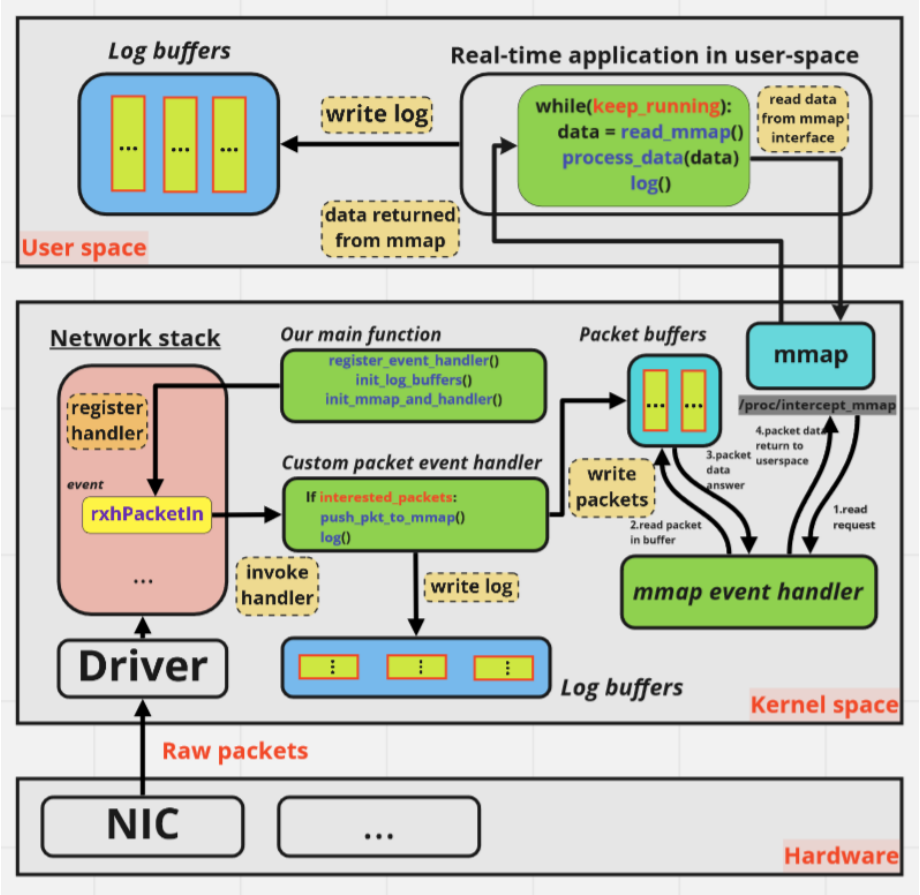


Figure 6.2: Design of kernel module method application. The incoming packet's flow is: arrive the network stack, intercepted by the *custom packet handler*, cloned and stored in the *packet buffer*, moved to *mmap memory* to be read by the user-space program. The original packet continues its journey after the *custom packet handler* had completed cloning.

The kernel program is responsible for:

- Intercepting packets in Linux network stack and performing real-time logic on desired packets. We can decide to let the packet continue its path, or drop it totally.

- Serving captured packets to user space program through shared memory (*mmap*). The endpoint is a *proc* file, located at */proc/intercept_mmap*. When the user-space program reads the *proc* file, the *mmap event handler* - which is part of our kernel module - will fetch the received packets from the buffer and return them in batch (usually 3-8 packets).

- Keeping logs. A timestamp is taken after the handler has verified the packet. The module stores timestamps in multiple arrays and exports them when the module is unloaded. A production program should be able to automatically export logs in

order to maintain continues operation, however this feature was dropped in our application due to time limitation.

The rest of the application live in the user space and are responsible for:

- Fetching packets from kernel module. This is done by calling the *read()* system call on the */proc/intercept_mmap* file. If there are packets available, a buffer filled with packet raw data will be return (usually 3-8 packets). Otherwise, buffer with only nulls is returned.
- Processing the packets. In our application, the user-space program only verifies the interested packets without doing any other processing. However, in a real-life application, this user-space program would have the same implementation and roles as a normal application, except how it retrieves the packets from the kernel program.
- Same as the kernel program, the user-space program also keeps logs. The module stores timestamp logs in multiple arrays and exports them when the module is unloaded.

6.2 Kernel Program

After the kernel module part had been injected into the kernel, it registers itself with an event (*netdev_rx_handler_register*). From now on, the module's packet-handle-logic "lives" in the kernel and will be called whenever a network packet arrives in the network stack. During the run time, logs will be written in buffers. When the kernel module is unloaded, the function *module_exit()* will be triggered to de-register the event, clean up memory and export log to file.

6.2.1 Module Initiation

In a modern Linux kernel module, the initiation process is called in the macro *module_init()*.

```
1   module_init(myinit)
2   module_exit(myexit)
3   MODULE_LICENSE("GPL");
```

The function *myinit* will be executed only once when the module is loaded to do the following jobs:

Allocate memory for logs and packet buffers: due to some limitations, each buffer should not exceed a value. We encountered several system crashes when we try to allocate big chunks of memory, so we limited the size of each buffer to 256KB. The used system call to allocate continues memory in kernel space is *__get_free_pages()*, which returns free pages if the OS can find them. For example, *PAGE_ORDER = 6* results in 2^6 pages * 4KB/page = 256KB. We use the plain *kmalloc()* system call in the program, however these usages can be safely replaced with helpers such as *kzmalloc()* that allocates and sets memory to zero, *kmalloc_array()* for array memory allocation, ...

```
1   /* Allocate memory for temporary buffer */
2   buff_from_here = (unsigned char *) kmalloc(PKT_BUFFER_SIZE*MAX_PKT, GFP_KERNEL);
3   /* Allocate memory for log buffers */
4   unsigned long *r;
5   for (int i=0; i<NUM_LOG_BUFF; i++) {
```

```
        r = (unsigned long *) __get_free_pages(GFP_KERNEL, PAGES_ORDER);
        if (!r) {
            // error
            printk(KERN_INFO "[RXH] ERROR: __get_free_pages");
            for (int j=0; j<i; j++) {
                free_pages((unsigned long) log_buffs[j], PAGES_ORDER);
            }
            return -1;
        }
        log_buffs[i] = r;
    }
```

Register the handler to the event:

The module uses netdev_rx_handler_register[5] system call to register the handler (covered in Section 6.2.2) to the receiver path of a network device (in our case the network device's name was *ens33*).

```
rx_handler_result_t rxhPacketIn(struct sk_buff **ppkt) {
    // logic
    // ...
    return RX_HANDLER_PASS;
}
// "device" is a network device, which has device->name == "ens33" on our system.
regerr = netdev_rx_handler_register(device,rxhPacketIn,NULL);
if (regerr) {
    printk(KERN_INFO "[RXH] Could not register handler");
} else {
    printk(KERN_INFO "[RXH] Handler registered with device [%s]\n", device->name);
}
```

6.2.2 Packet handler

We implement the logic that will be applied on each packet in this function, namely filtering interest packets and storing them in buffer.

```
/* rxhPacketIn function will be called when a packet is received on a registered netdevice */
rx_handler_result_t rxhPacketIn(struct sk_buff **ppkt) {

    struct sk_buff* pkt;
    struct iphdr *ip_header;
    struct udphdr *udp_udphdr;

    pkt = *ppkt;
    ip_header = (struct iphdr *)skb_network_header(pkt);

    /* Check if IPv4 packet */
    if(ip_header->version != 4){
        // printk(KERN_INFO "[RXH] Got packet that is not IPv4\n");
        return RX_HANDLER_PASS;
    }

    /* If not UDP, pass the packet */
    if(ip_header->protocol != 17) {
        return RX_HANDLER_PASS;
    }

    /* Parse UDP header */
    udp_udphdr = (struct udphdr *)skb_transport_header(pkt);
    if ((unsigned int)ntohs(udp_udphdr->dest) != 8080) {
        return RX_HANDLER_PASS;
    }

    u64 now = ktime_get_ns();
    unsigned long uid = 0;

    /* Write timestamp to log */
    memcpy(&uid, pkt->data+28+8, sizeof(unsigned long));
    if (uid<MAX_LOG_ENTRY && uid>=0) {
```

[5] https://www.kernel.org/doc/htmldocs/networking/API-netdev-rx-handler-register.html

```
34              unsigned int buff_index = uid / MAX_ENTRIES_PER_LOG_BUFF;
35              log_buffs[buff_index][uid % MAX_ENTRIES_PER_LOG_BUFF] = now;
36          }
37
38          /* Write data into the module's cache */
39          unsigned int try = 0;
40          unsigned int write_index = current_index;
41          void *pos = 0;
42          while (try<KM_FIND_BUFF_SLOT_MAX_TRY) {
43              if (status[write_index] == 0) {
44                  pos = buff_from_here + write_index*PKT_BUFFER_SIZE;
45                  // Copy packet (14 ethernet already tripped off till this stage) 20 ip, 8 udp
46                  memcpy(pos, pkt->data+28, (unsigned int)ntohs(udp_udphdr->len) - 8);
47                  // Set ks_time_arrival_2 to "now"
48                  status[write_index] = 1;
49                  current_index = (write_index + 1) % MAX_PKT;
50                  break;
51              }
52              write_index = (write_index + 1) % MAX_PKT;
53              ++try;
54              /* No free slot found. We will just discard the packet.
55                 */
56              if (try == KM_FIND_BUFF_SLOT_MAX_TRY) {
57                  count_pkt_overflow += 1;
58                  printk(KERN_INFO "Packet uid[%lu] cannot be written in cache: No free slot!", uid);
59              }
60          }
61          return RX_HANDLER_PASS;
62      }
```

- Handler function will be called whenever the registered event is fired. The function receives a *sk_buff* pointer that points to the packet in the network stack's buffer. In a real-life application, the handler is supposed to apply real-time logic on the packets, and then return the action code to the network stack to discard or let the packet travel its path.
- We search only for the needed packets - in this case the UDP packets that entered *DEST_PORT 8080*. These packets will be cloned and stored in packet buffer.
- Also the arrival time is recorded and written in the log.
- At the end of our function, *return RX_HANDLER_PASS* means we let the network stack handle the packet as normal.

6.2.3 Module Unload

The function *myexit* has the following tasks:

De-register the packet handler with kernel: similar to the registration, de-registration is done through a system call: *netdev_rx_handler_**unregister**(device)*

Cleanup: Depends on how the buffer was allocated, it will be freed using either *kfree()* or *free_pages*.

Export log files: to simplify the process, we write the content in the log buffers into files using *vfs_write()* system call. This is however not recommended in production.

```
1   /* write log buffers in file */
2   unsigned long zeroed = 0;
3   printk(KERN_INFO "[RXH] Kernel module exporting log");
4   struct file *fp;
5   mm_segment_t fs;
6   loff_t pos_file;
7   fp = filp_open(path_km, O_RDWR | O_CREAT, 0777);
8   if (IS_ERR(fp)) {
9       printk("create file error\n");
10      return;
```

```
    }
    fs = get_fs();
    set_fs(KERNEL_DS);
    pos_file = 0;
    char temp[128];
    for (unsigned long i=0; i<MAX_LOG_ENTRY; i++) {
        memset(temp, '\0', 128);
        unsigned int buff_index = i / MAX_ENTRIES_PER_LOG_BUFF;
        snprintf(temp, 100, "%lu\n", log_buffs[buff_index][i % MAX_ENTRIES_PER_LOG_BUFF]);
        vfs_write(fp, temp, strlen(temp), &pos_file);
        if (log_buffs[buff_index][i % MAX_ENTRIES_PER_LOG_BUFF] == 0)
            zeroed += 1;
    }
    filp_close(fp, NULL);
    set_fs(fs);
```

6.3 User-space Program

The user-space program part can be started once the kernel module is loaded. The program needs to allocate memory for log and buffer, open the *proc* file and mount it using *mmap* system call. From now on, the program can start pulling data from the kernel part by reading this *proc* file description.

In a more complete application, the user space program should have capability to manage the kernel module program, for instance: load and unload program, statistic gathering, ... These features are skipped in our application to reduce development time and effort.

6.3.1 Program Initiation

Allocation memory for logs and buffer: We use standard *malloc()* system call to allocate memory.

```
/* Allocate memory for log buffers */
unsigned long *r;
for (int i=0; i<NUM_LOG_BUFF; i++) {
    r = (unsigned long *) malloc(MAX_ENTRIES_PER_LOG_BUFF*8);
    if (!r) {
        // error
        for (int j=0; j<i; j++) {
            free(log_buffs[j]);
        }
        return -1;
    }
    log_buffs[i] = r;
}
printf("Allocated NUM_LOG_BUFF[%d] for MAX_LOG_ENTRY[%d] packets!\n",
        NUM_LOG_BUFF, MAX_LOG_ENTRY);
```

Setup mmap: mapped memory is a common method used to exchange data between kernel space and user space. Calling *mmap* with *PROT_READ* (memory protection of the mapping) flag to signal read permission for the application [t18]. Because of this privileged protection, user-space program must be run with *sudo* to read the data (usually with the *read* system call) from this memory area.

```
/* Open proc file */
// name_buff in our application is /proc/intercept_mmap
fd = open(name_buff, O_RDWR | O_SYNC);
if (fd < 0) {
    perror("open");
    assert(0);
}
```

```
/* Map mem */
page_size = sysconf(_SC_PAGE_SIZE);
address1 = mmap(NULL, page_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (address1 == MAP_FAILED) {
    perror("mmap");
    assert(0);
}
```

6.3.2 Read packet from kernel module and main logic

```
while(keep_running) {
    memset(buff_read, '\0', BUFFER_SIZE);
    /* ssize_t r = */pread(fd, buff_read, BUFFER_SIZE, 0);
    for (int i=0; i<PKTS_PER_BUFFER; i++) {
        if (buff_read[i*PKT_BUFFER_SIZE] == '\0')
            continue;
        uint64_t now = get_nsecs();
        // Extract data from packet
        struct Payload pl;
        memcpy(&pl, buff_read+i*PKT_BUFFER_SIZE, sizeof(struct Payload));
        /* Check packet */
        if (pl.client_uid!=0 && pl.type==PL_DATA && pl.created_time!=0) {
            // Add timestamp to log at uid
            if (pl.uid <MAX_LOG_ENTRY && pl.uid >= 0) {
                unsigned int buff_index = pl.uid / MAX_ENTRIES_PER_LOG_BUFF;
                log_buffs[buff_index][pl.uid % MAX_ENTRIES_PER_LOG_BUFF] = now;
            } else {
                printf("Something␣wrong␣[%lu]", pl.uid);
            }
        }
    }
}
```

- This *while* loop will busy pull the kernel module for new data. To improve efficient, combine with *usleep()* after each read, or use *poll()* system call.
- *pread(fd, buff_read, BUFFER_SIZE, 0)* reads *BUFFER_SIZE* bytes from the file description *fd* into buffer.
- After verifying the packet, the main logic of application can be called.

6.3.3 Program Termination

Same as to kernel program, log must be exported to file and memory must also be freed before the program terminates.

Cleanup: Memory for logs and buffer will be freed using standard *free()*.

Export log: Flush data from logs buffer to file using standard *fprintf()*.

CHAPTER 7

eBPF APPLICATION IMPLEMENTATION

7.1 Introduction

7.1.1 Overview

eBPF was designed for BSD (hence the name Berkeley Packet Filter, BPF) to filter packet using predefined rules. Unlike kernel module that shares the same memory space with the kernel, BPF runs custom program - or *filter* as called back then - **on event** in a sand-boxed RISC virtual machine. It was ported to Linux and then extended with more features and improvements (hence the *e*), such as moving to 64-bit registers and increasing the number of registers and more [m2]. Recently, thanks to the valuable performance improvements and exposing eBPF to the user-space, eBPF can be finally used for purposes such as tracing or profiling by end-user . There are many currently supported events in different parts of the system, i.e network packets, tracing events, classification events, ... and there are more planned in the future [t17].

An eBPF application can have one or more kernel programs and optionally user-space programs. In our example application, the kernel program will be inserted into

the Linux network driver level. The user-space program runs in the user-space and communicates with the eBPF kernel program through special mechanism.

7.1.2 Technical details

An eBPF kernel program is written in a subset of the C-programing language (without loops, global variables, variadic functions, floating-point numbers, and passing structures as function arguments) and will be compiled using clang/LLVM - with *gcc* might be able to support eBPF target in the future - to produce eBPF binary program. On supported platform (x86-64, ARM32 and 64, MIPS 64 ...), the kernel's just-in-time compiler can translate eBPF code into platform's bytecode for additional performance.

Unlike kernel module, eBPF has a much more modern mechanism for communication between kernel programs and user-space programs, called *BPF map*. This map is a data structure abstraction that implements a key-value map, which supports a wide range of types of data and functional modes for different purposes. For example, *BPF_MAP_TYPE_HASH* is a hash table and *BPF_MAP_TYPE_PERCPU_HASH* is a per-CPU hash table, which means each CPU core has an own hash map for storing data.

Be designed to be safe, the kernel has a verifier to check the program for possible risks, such as *loop*, unreachable code, un-allowed system calls, reference to removed maps, ... Depending on the type of the eBPF kernel program, a subset of the eBPF's system calls is available to the program, such as program of type *BPF_PROG_TYPE_SOCKET_FILTER* has access to *bpf_map_lookup_elem(), bpf_map_update_elem(), ...* [t17].

A modified, simple eBPF kernel program from the XDP tutorial[1] repository has the following code:

```
1  /* SPDX-License-Identifier: GPL-2.0 */
2  #include <linux/bpf.h>
3  #include <bpf/bpf_helpers.h>
4
5  struct bpf_map_def SEC("maps") xdp_stats_map = {
6      .type = BPF_MAP_TYPE_PERCPU_ARRAY,
7      .key_size = sizeof(int),
8      .value_size = sizeof(__u32),
9      .max_entries = 64,
10 };
11
12 SEC("xdp_sock")
13 int xdp_sock_prog(struct xdp_md *ctx)
14 {
15     int index = ctx->rx_queue_index;
16     __u32 *pkt_count;
17
18     pkt_count = bpf_map_lookup_elem(&xdp_stats_map, &index);
19     if (pkt_count) {
20         if ((*pkt_count)++ & 1)
21             return XDP_PASS;
22     }
23     return XDP_PASS;
24 }
25
26 char _license[] SEC("license") = "GPL";
```

- A typical eBPF kernel program has the *rules* defined in the function with the macro *SEC("xdp_sock")*. This marks the attribute *xdp_sock* in the ELF binary

[1] https://github.com/xdp-project/xdp-tutorial/tree/master/advanced03-AF_XDP

output file. *SEC("maps")* is used to marks the map structures. In this example, the map's name is "xdp_stats_map" and it can hold 64 integer values.

- eBPF kernel program must be licensed under GPL, otherwise it will be rejected from loading. The standard line *char _license[] SEC("license") = "GPL"* is placed at the end of the program.
- This kernel program is compiled using the command *clang -O2 -Wall -target bpf -c xdp-example.c -o xdp-example.o* and loaded using *sudo ip link set dev ens33 xdp obj xdp-example.o sec xdp-sock*. In our application, the kernel program can be inserted from the user-space program to automate the process.

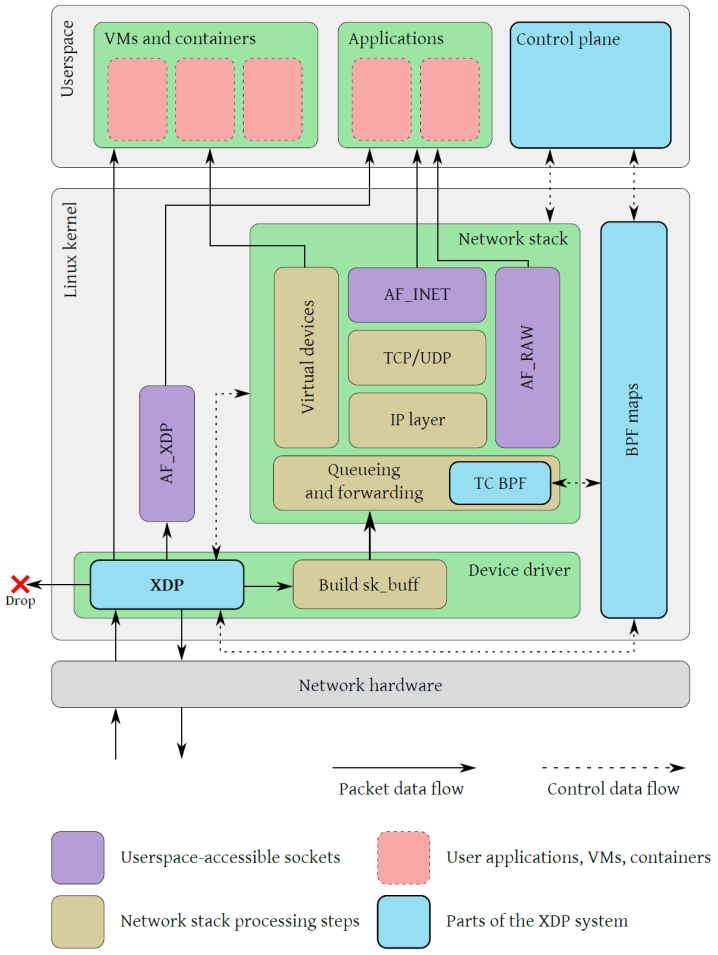


Figure 7.1: XDP's integration with the Linux network stack [p14]

eBPF has been widely used among the kernel developer community. In the user-space, perhaps XDP socket (AF_XDP) is the most famous example of eBPF application (Figure 7.1), which was a huge success and has great potential for high-performance networking application [p17] [p14].

Initially, we intent to use XDP socket in the application, however this will disable the packet inception capability of the kernel module application and thus will not allow us to run both example programs simultaneously for direct comparison. Switching to plain eBPF program didn't fix the problem either, albeit the attempt has already taken too much time and therefore we ended up with the eBPF example application as will be introduced afterwards.

7.1.3 Architecture of our eBPF application

Our example eBPF application has 1 kernel program, 1 user-space program and uses 2 maps for log (*uid_timestamps*) and data packet (*pkt_payload*). There is also a *xdp_stats_map* used for solely debugging purpose.

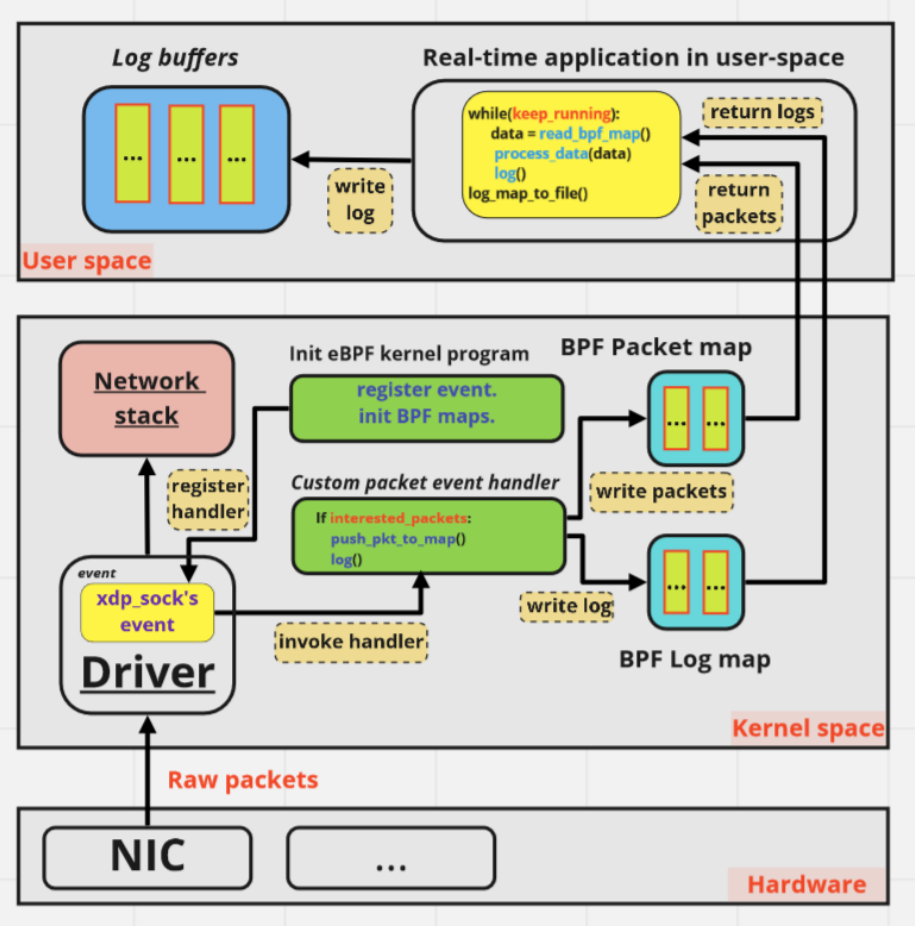


Figure 7.2: Design of eBPF method application. An incoming packet arrives in the driver's buffer, intercepted by eBPF kernel program, cloned and stored in the *BPF packet map*, which will be read by the user-space program. The kernel program lets the packet free after it has stored the cloned packet into the packet map.

The kernel program is responsible for:

- Intercepting each packet arrived in Linux network stack, let the uninteresting packets continue. Storing a clone in *packet map*. We can decide to let the packet continue its path, or drop it totally.
- Keeping logs. A timestamp is taken after the handler had verified the packet. Unlike kernel module program, eBPF kernel program doesn't manage the memory for log buffers manually, but stores the logs in a *bpf map*. User-space program is responsible for reading and writing logs to file when the application terminates.

The rest of the application live in the user space and are responsible for:

- Fetching packets from kernel module by reading *bpf packet map*. We use *busy pulling* which uses a while loop to keep reading data from map, however an implementation using *poll()* system call could have reduced the system load significantly.
- User space program keeps and exports log to log files. It also handles log exporting for the kernel program by reading the log map and writing to files.

7.2 Kernel Program

7.2.1 Program Initiation

Dissimilar to the kernel module program, eBPF kernel program exchanges data with other programs using *bpf map*. This reduces our workload significantly and eliminates the need for buffer allocation.

Define bpf maps: 2 maps need to be defined for storing log and packet data. Note that the max map length we are able to request is around 10^6 entries, correct number is for us currently unknown.

```
struct bpf_map_def SEC("maps") uid_timestamps = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(unsigned int),
    .value_size = sizeof(unsigned long),
    .max_entries = 1 + MAX_LOG_ENTRY,
};

struct bpf_map_def SEC("maps") pkt_payload = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(unsigned int),
    .value_size = sizeof(struct Payload),
    .max_entries = 1 + MAX_LOG_ENTRY,
};
```

Register the handler to the event: This is done implicitly in user-space program as in our application, or explicitly using command line, i.e:

```
sudo ip link set dev ens33 xdp obj ebpf_measure_kern.o sec xdp_sock
```

The packet handler is defined as follow:

```
    /* Parsing header */
    ip_type = parse_iphdr(&nh, data_end, &iphdr);
        if (ip_type != IPPROTO_UDP) {
        return XDP_PASS;
        }
    if (parse_udphdr(&nh, data_end, &udphdr) != sizeof(struct Payload)) {
        return XDP_PASS;
    }
    if (udphdr->dest != bpf_htons(DEST_PORT)) {
```

```
10              return XDP_PASS;
11          }
12          /* End parsing */
13
14          /* Process packet payload */
15          if (data_end >= data + 106) {
16              if (data_end >= data+42+sizeof(struct Payload)) {
17                  unsigned long now = bpf_ktime_get_ns();
18                  struct Payload *pl;
19                  pl = data+42;
20                  unsigned long *time_in_map = bpf_map_lookup_elem(&uid_timestamps, &pl->uid);
21                  if (time_in_map) {
22                      *time_in_map = now;
23                  }
24                  /* Copying payload data*/
25                  int r = bpf_map_update_elem(&pkt_payload, &pl->uid, pl, BPF_ANY);
26                  if (r==-1) {
27                      bpf_printk("Failed add to pkt_payload uid[%lu]: ", pl->uid);
28                  }
29                  /* Counting */
30                  pkt_count = bpf_map_lookup_elem(&xdp_stats_map, &index);
31                  if (pkt_count) {
32                      (*pkt_count)++;
33                  }
34                  return XDP_PASS;
35              }
36              return XDP_PASS;
37          }
38          return XDP_PASS;
39      }
```

- Header parsing and examining: we check the packet's size, port, protocol.
- We clone the interested packet and store the copy in the packet map. Other business logics should be implemented from this position, i.e drop packets, immediately answer, ...
- The program also takes timestamp and stores in log map.
- Decision about the fate of the original packet. There are pre-defined return-codes to signal the BPF VM about the next action, i.e *XDP_DROP* to drop this packet, *XDP_PASS* to let packet be handled by network stack [t14].

7.3 User-space Program

Besides the basic functionalities as in kernel module use-space program, injecting kernel program is also handled here using function *load_bpf_and_xdp_attach()* [t30] to XDP event.

7.3.1 Program Initiation

Load kernel program and find bpf maps:

```
1   bpf_obj = load_bpf_and_xdp_attach(&cfg);
2   if (!bpf_obj) {
3       /* Error handling done in load_bpf_and_xdp_attach() */
4       exit(EXIT_FAILURE);
5   }
6
7   /* Time Log map */
8   time_map = bpf_object__find_map_by_name(bpf_obj, "uid_timestamps");
9   time_xsks_map_fd = bpf_map__fd(time_map);
10  if (time_xsks_map_fd < 0) {
11      fprintf(stderr, "ERROR: no xsks map found: %s\n",
```

```
12              strerror(time_xsks_map_fd));
13          exit(EXIT_FAILURE);
14      }
15
16      /* Pkt map */
17      pkt_map = bpf_object__find_map_by_name(bpf_obj, "pkt_payload");
18      pkt_map_fd = bpf_map__fd(pkt_map);
19      if (pkt_map_fd < 0) {
20          fprintf(stderr, "ERROR: no xsks map found: %s\n",
21              strerror(pkt_map_fd));
22          exit(EXIT_FAILURE);
23      }
```

Allocate buffers for log: we use standard *malloc()*.

Fetch and process packets from bpf map: Similar to kernel module application implementation, we use a while loop to keep fetching captured packets. This reading from map is done by calling *bpf_map_lookup_elem()*.

```
1   static bool fetch_and_process_packet(unsigned int next_uid)
2   {
3           struct Payload map_pl; //map payload
4           if (bpf_map_lookup_elem(pkt_map_fd, &next_uid, &map_pl) == 0) {
5                   if (map_pl.client_uid!=0 && map_pl.type==PL_DATA && map_pl.created_time!=0) {
6                           unsigned long now = gettime();
7                           if (map_pl.uid <MAX_LOG_ENTRY && map_pl.uid >= 0) {
8                                   unsigned int buff_index = map_pl.uid / MAX_ENTRIES_PER_LOG_BUFF;
9                                   log_buffs[buff_index][map_pl.uid % MAX_ENTRIES_PER_LOG_BUFF] =
                                            now;
10                                  return true;
11                          }
12                  }
13          }
14          return false;
15  }
```

7.3.2 Program Termination

Cleanup and export log data:

Cleanup: Memory for logs will be freed using standard *free()*.

Export user-program log: Flush data from logs buffer to file, using standard *fprintf()*. Reuse implementation from kernel module application.

Export kernel-program log and remove kernel program: The user space program first reads and exports all log entries from log map, then removes the kernel program from kernel space.

```
1   /* Export log file for kernel space (from map) */
2   FILE *fp2;
3   fp2 = fopen(path_ebpf_kern, "w");
4   unsigned long v = 0;
5   for (unsigned i=0; i<MAX_LOG_ENTRY; i++) {
6       if (bpf_map_lookup_elem(time_xsks_map_fd, &i, &v) == 0) {
7           fprintf(fp2, "%lu\n", v);
8           // if (v!=0) {
9               // printf("uid[%d] ks[%lu]\n", i, v);
10          // }
11      } else {
12          printf("Error reading from time log map with bpf_map_lookup_elem!\n");
13      }
14  }
15  printf("Zeroed: %lu\n", zeroed);
16  fclose(fp2);
17
18  /* Cleanup */
19  xdp_link_detach(cfg.ifindex, cfg.xdp_flags, 0);
```

CHAPTER 8

EVALUATION

This chapter will walk through the process and outcome of the evaluation phase. Section 8.1 introduces the test setup and test cases, in which the example applications will be run and data will be collected. Section 8.2 presents the test results for each test case, and our observation in Section 8.3. Finally, the Section 8.4 will analyze the results against the defined approach's requirements.

8.1 Introduction

This chapter presents the performance evaluation of the proposed approach in relation with the requirements defined by the Chapter 3: Requirements. The performance of the example application will be placed in comparison with a normal Linux socket server, and in-directly compares the *kernel module* and *eBPF* methods.

8.1.1 Overview

In an ideal test setup, the server will have all 3 applications (Linux socket, eBPF- and kernel-module based) running together to ensure 1. fairness and 2. allow direct comparison between the methods. Unfortunately, the kernel module and eBPF method

applications couldn't run simultaneously on the same system due to an unsolved technical issue that causes packets to flow unnoticeably by the Kernel Module. Instead of investigating further into solving the problem (which in fact, had resulted in failure), we decided to **test each of these 2 example applications with the Linux socket application** to demonstrate how effective the methods are over Linux socket, and use their improvement over Linux socket application for indirect-comparison between the 2 methods, or in other words, use Linux socket application as the base line.

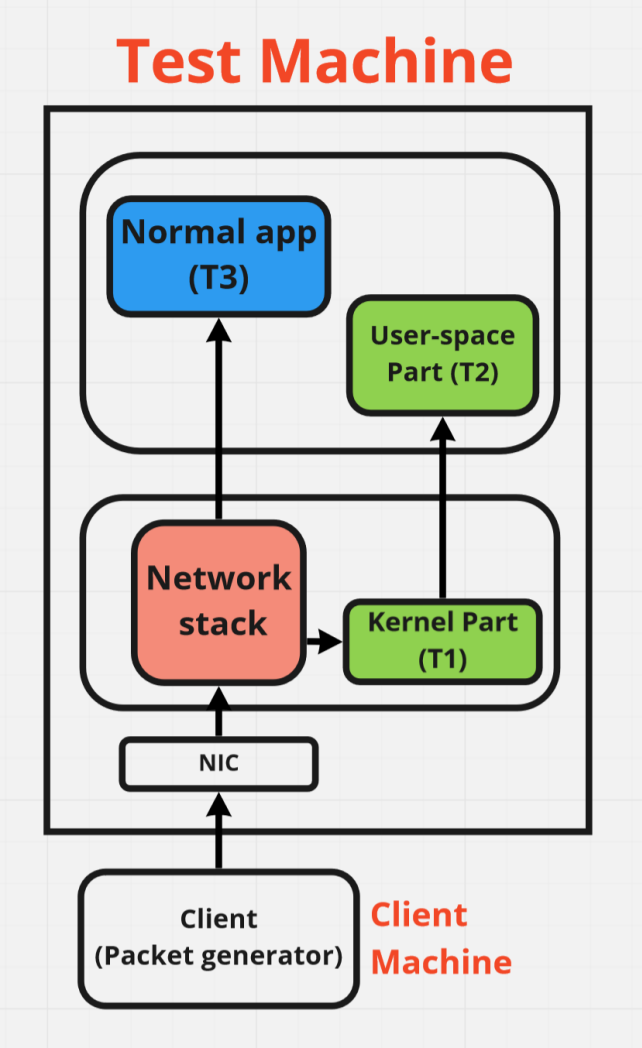


Figure 8.1: Test architecture: client machine is the packet generator that sends network packet to the test machine (called *run-box* machine). The packets will then flow to the network stack, where they will be intercepted, processed, cloned and transmitted (by the kernel part of the example application) to the user-space part. The kernel part is where the first measurement takes place (T1). The user-space part is expected to receive the packet shortly after (T2 moment). Lastly, the original packets will arrive at the normal app at T3 time.

The test machine that runs the server applications is a VM Ubuntu Desktop box, virtualized by VMware Workstation Player[1] on a Windows 10 host. The client runs on a compact VM box while the VM for servers has excess resources and is as powerful as a standard workstation (the VM has higher single core score, but multi-core performance is on-pair with a Intel Xeon E3-1230 v5@3.40GHz), as reported by Passmark[2] benchmark:

Listing 8.1: Passmark reports specification of VM for server applications

```
1 OSName: Ubuntu 20.04.2 LTS
2 Kernel: 5.8.0−63−generic
3 Processor: Intel(R) Core(TM) i9−9900K CPU @ 3.60GHz
4 NumCores: 4
5 CPUFrequency: 3600
6
7 CPU_SINGLETHREAD: 2976.5547227679667
8 SUMM_CPU: 7777.4837146794143
```

8.1.2 Test cases in details

Each test needs to run 2 sub-tests, which constructs a **single test**: sub-test-1 has *Kernel Module and Linux socket application* running on the same server, and in sub-test-2 the server will have *eBPF and Linux socket application*. A collection of 5-runs of a **single test** comprises a single **Test case**. Different test cases have different parameters:

- Number of packets in total: 524.288, or 65.536, or 32.768 pkt.
- Size of payload in each packet (in Byte): 64, or 512, or 1024 byte.
- Sending rate in packet/sec: 50.000, or 6.000, or 3000 pkt/s.
- System load: Non-stress (all cores idle), or Stress (all cores 100% load, using *stress –cpu $(nproc)*).

8.2 Performance Test

In the test phase, each test is assigned with a *case* that defines the testing condition. We adjust the number of packets, sizes and sending rates so 1. The test machine will not be overwhelmed and become unstable and 2. Client sends the same amount of data (in size) to the server, which yields 32MB in our case. So, in total there are: 6 **Test cases** (stress and non-stress) * 5 Repeats = 30 **Single tests**. Each **Single Test** consisted of 2 sub-tests for eBPF and Kernel Module application, makes it in total 60 test-runs.

8.2.1 Single test result

The following graphs show the ***distribution of latencies between the kernel-space program and user-space programs*** (using Linux socket or our custom methods), **in each test case** (combined 5 test-runs together). These latencies are calculated by subtracting the timestamp of events: packet first arrives in kernel program (**T1**), while **T2** and **T3** reflect arrival time of the very same packet in the user-space part of our example server and Linux socket server, respectively. We assigned the name T2 and

[1] https://www.vmware.com/products/workstation-player/workstation-player-evaluation.html
[2] https://www.passmark.com/products/performancetest/

T3 for the example server and Linux socket server because our example servers are expected to receive the packets earlier than the traditional Linux socket server could (hence the order).

The x-axis displays the latency ranges in microseconds and the y-axis indicates the number of packets that have their latency values belong to a particular range. Each individual graph shows *how the latency value ranges (in microsecond) - observed in the tests and calculated using the kernel-program timestamp as base (T1) - distribute*. These distributions describe several performance characteristics of each server application, such as mean, median, encounter probability ... The **homogeneous color bars represent the distribution of T2-T1** ranges while the **patterned bars show T3-T1** ranges. A pair of plain and patterned bar for each range provides side-by-side comparison. Because of the deviation in distribution, the y-axis is set to logarithmic-scale to save vertical space (Section 8.2.2 has both linear-scaled and logarithmic-scale y-axis versions, which show the enormous differences). A relative higher bar can therefore express a significant difference, for example the difference between 10e4 and 10e3.9 is over 20,000. It is expected to observe the latency value for most of the packets to be between 0-5 microseconds for our example application while Linux socket application has much higher values cross over a wider range, with most of them scatter between 5-70 microseconds. With T1 time being the base-time, this would mean:

- Our approach can deliver packet to the user-program in less than 5us in most cases.
- Linux socket server receives most of the packets after 5-70us, which is up to 65us later.
- The extreme values (greater than 70us) and their appearance frequencies play deciding role in a real-time system's quality of service. These plots show significant better test results with our applications.

Test: Number of packets: 32.768, Size: 1024 Bytes, Rate: 3.000 pkt/s

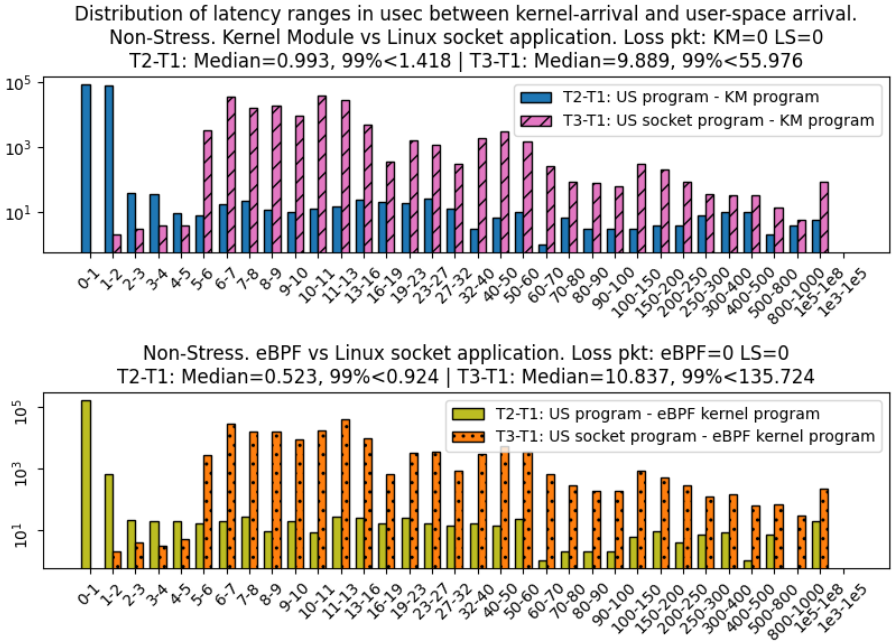


Figure 8.2: Non-stress, 32k packets, Rate 3kpps, Size 1024B

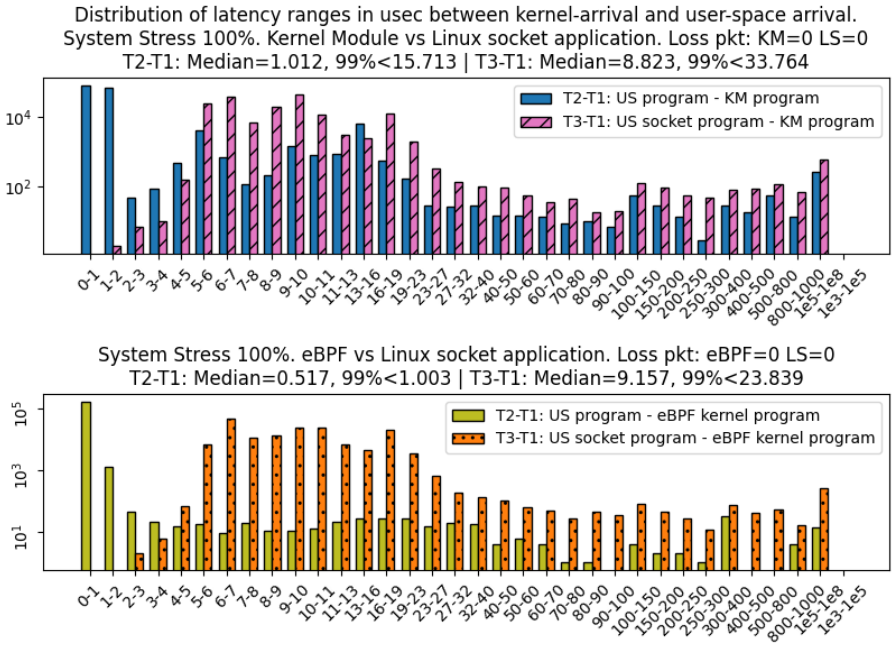


Figure 8.3: Stress, 32k packets, Rate 3kpps, Size 1024 bytes

Number of packets: 65.536, Size: 512 Bytes, Rate: 6.000 pkt/s

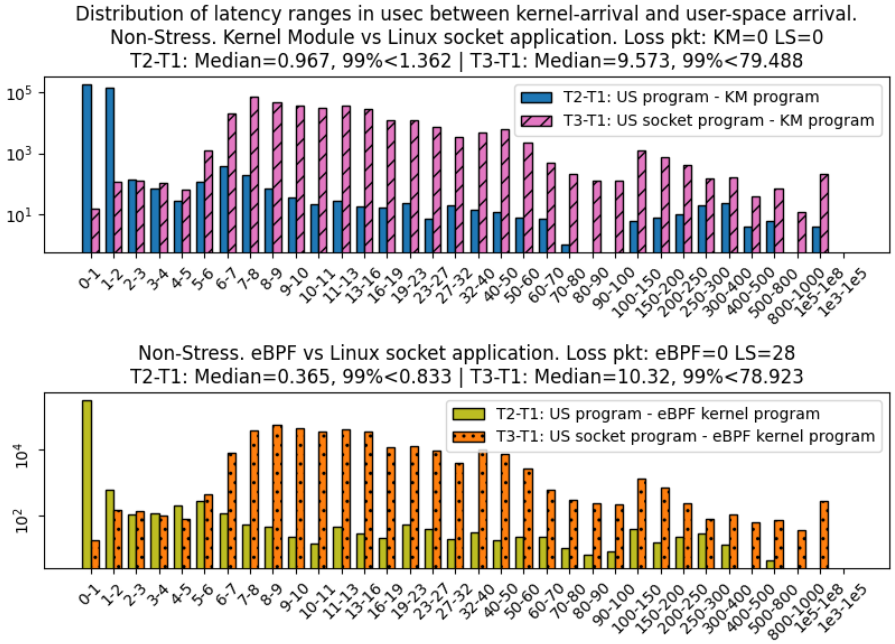


Figure 8.4: Non-stress, 65k packets, Rate 6kpps, Size 512 bytes

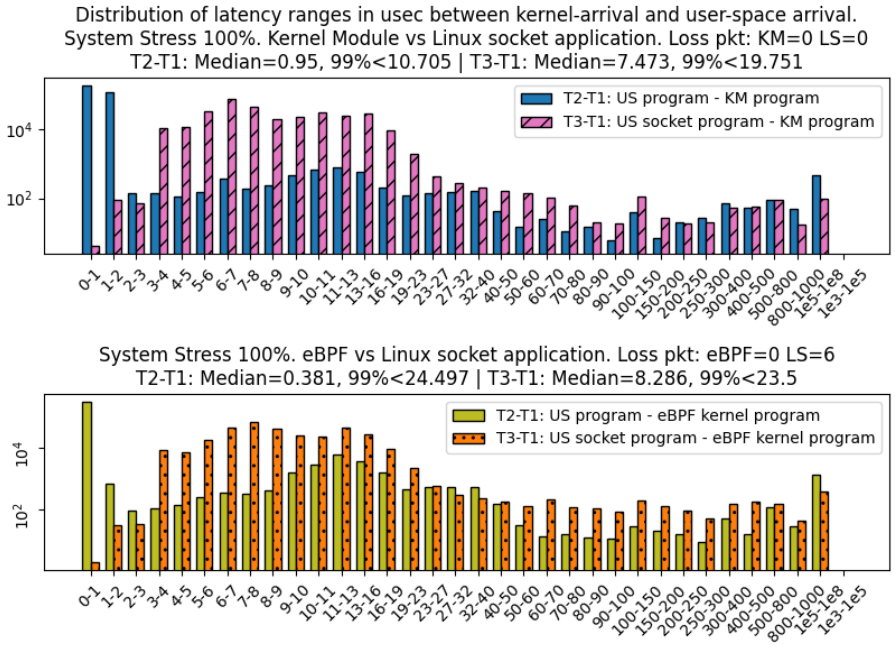


Figure 8.5: Stress, 65k packets, Rate 6kpps, Size 512 bytes

Number of packets: 524.288, Size: 64 Bytes, Rate: 50.000 pkt/s

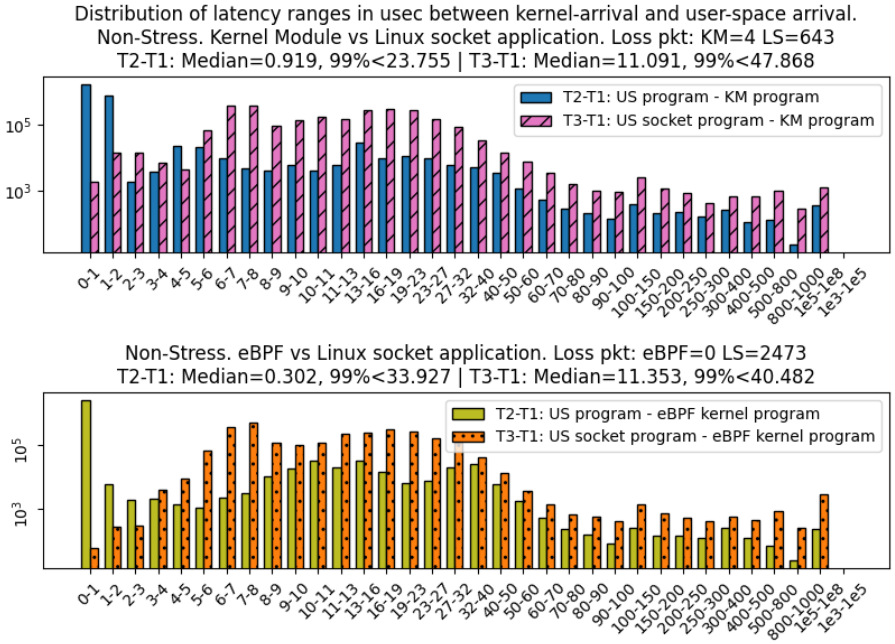


Figure 8.6: Non-stress, 524k packets, Rate 50kpps, Size 64 bytes

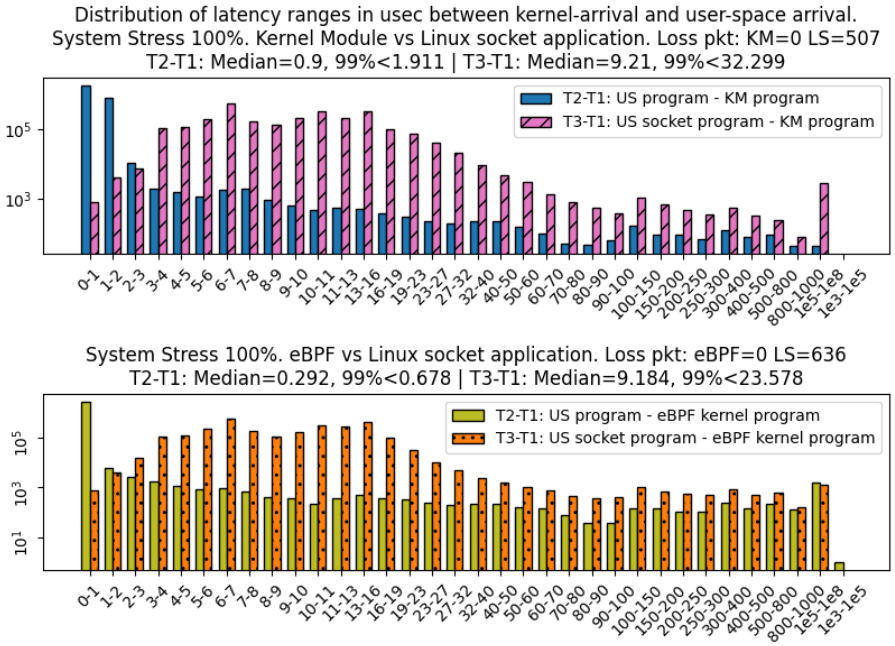


Figure 8.7: Stress, 524k packets, Rate 50kpps, Size 64 bytes

8.2.2 Congregated result

Results from all tests will be gathered together and plotted:

Combined result of stress and non-stress tests, in logarithmic scale

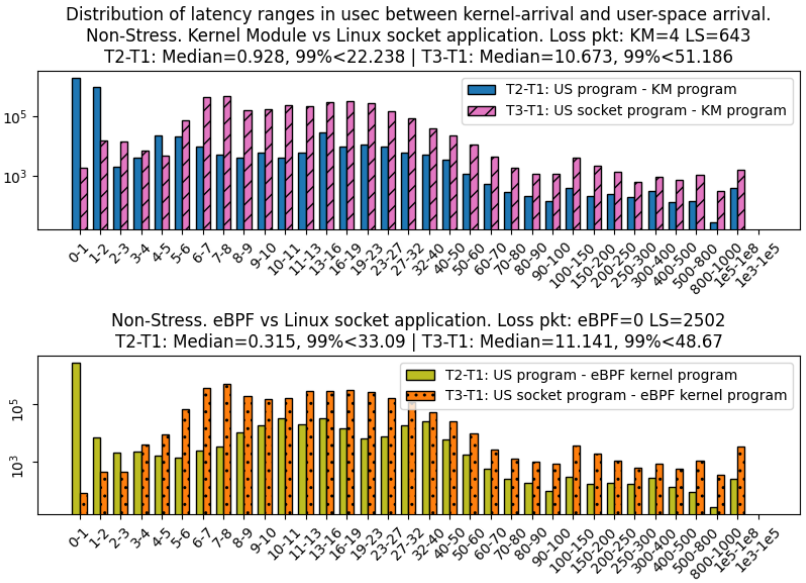


Figure 8.8: Non-stress, Combined result

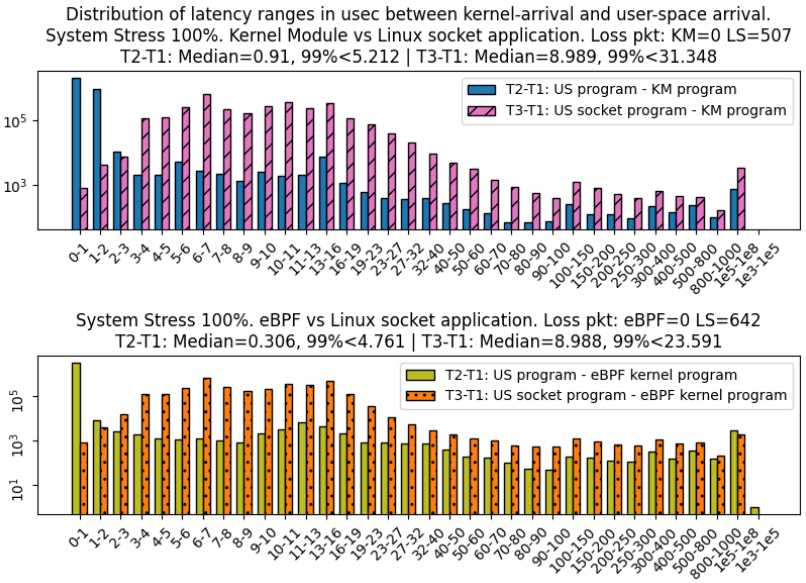


Figure 8.9: Stress, Combined result

Combined result of stress and non-stress tests, in linear scale

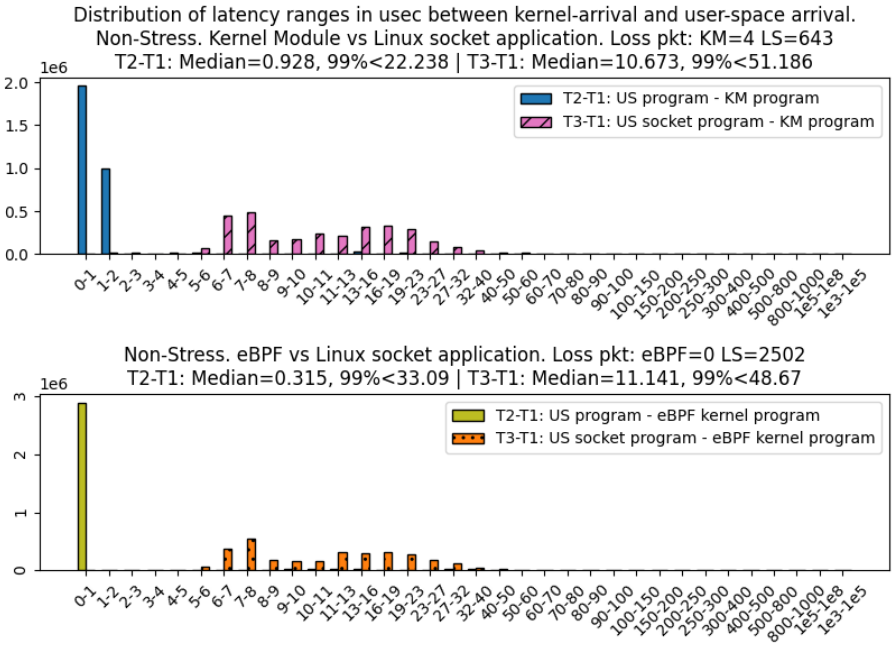


Figure 8.10: Non-stress, Combined result

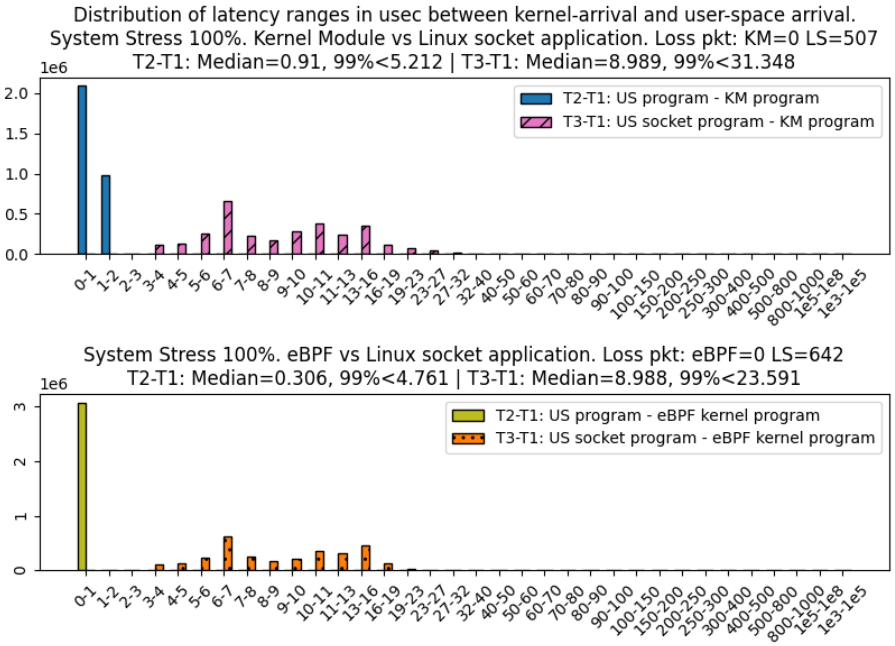


Figure 8.11: Stress, Combined result

8.3 Observational Validation

Overall, the approach offers huge performance boost over traditional Linux's network socket, both in lower average latency value and lower number of high latency value. The approach is superior in handling great number of packets (high-rate packet receiving) as be seen in Figure 8.7 and Figure 8.6: the socket servers suffer hundreds of packet loss cases compared to only 4 cases with the kernel-module application.

Stress test has negative impacts on the server's performance: higher absolute values, and increase in number of such incidents. More over, high system load might lead to more packet loss incidents which we acknowledged during developing, although couldn't distinguishably produce in the test phase. The stress test also reduces the difference between the Linux socket and our custom server, results in the **lower** median and 99% values of the stress test cases compared to the respective non-stress cases. This is because these differences express *the performance improvement of in-kernel method over the Linux socket*. The extra load caused by the *stress* command reduces these distances, however the stock network socket is still left far behind by the in-kernel processing approach as showed linearly in Figure 8.10 and Figure 8.11.

Further performance comparisons:

- Both eBPF and Kernel Module method are able to deliver the majority of network packets in shorter time (smaller latency) than Linux socket. The tests show 10x improvement in median values, which don't exceed 1 microsecond for most cases over traditional Linux socket. There are also 99% of the packets were delivered faster than Linux socket server, varied from 2-10 times.
- Packets arrived to user-space program of our example application between less than 1 to 5 microseconds while the traditional Linux socket could only fetch the packets after 5-70 microseconds.
- There are packets with extreme latency values well over 100 microseconds. Using the proposed approach can however lower the number of these incidents up to 100 times.
- During the test, Linux socket server sustained packet drop with the number of loss packets range from 0 to over 2400. We only acknowledged the kernel module application failed to fetch 4 packets compared to 643 by the Linux socket server in the same test case (Figure 8.6).
- The example applications used a *while* loop to pull packets from the kernel programs and therefore pushed CPU usage to 100% on one CPU core, compared to almost zero CPU usage by the Linux socket server which proved the efficient of the Linux's implementation. This can be improved by optimizing the programs, splitting the load to multiple cores, or using less aggressive packet-pulling methods, such as adding a short sleep time between fetches or utilizing the Linux's *poll* system call (as used in the XDP tutorial example application, which we acknowledged to have an identical low CPU usage pattern).
- The example applications had small memory footprint. The packet buffers required no more than a few kilobytes of memory while log buffers might take a few megabytes to log all events, however this kind of log implementation is irrelevant

to production purposes. All tested applications are fairly simple and therefore have no significant difference in size or memory usage.

8.4 Requirements Evaluation

This section will evaluate the result against the pre-defined requirements and other approaches's performance.

Table 8.1: Requirements in categories as stated in Chapter 3

Research target	**Requirement detail**	Code
Functional requirements: Determining approach ability	Assess performance improvement	R1A
	Assess application's compatibility	R1B
	Viable in real-world application?	R1C
Business requirements: Comparison performance	Compare to Linux performance	R2A
	Compare to other approaches performance	R2B
	License	R2C
	Technical Support	R2D

.

8.4.1 Functional requirements evaluation

From the obtained results, we conclude that the approach has fulfilled all the defined requirements, as explicated below:

Performance improvement over Linux's default capability (R1A):

- **Lower latency than vanilla kernel** (reacting to events) OR **Lower processing time than vanilla kernel** (through more CPU time, less de-scheduled, less system calls ...)
- **Lower jitter, more predictable than vanilla kernel**: distribution of processing time values should be close to average values.

Through the tests it is clear that the approach can improve the Linux's real-time performance for networking purpose by 1. reducing average packet's latency by a factor between 2-10 times and 2. offering lower number of spikes in latency. This would result in a smoother experience for the user-space program and enhance Linux's real-time capability by providing a *faster* (regarding reaction time) and *more stable* foundation for real-time tasks. Besides, because the kernel part can react to event much earlier than an ordinance user-space application, the application is able to process certain tasks even more quickly (such as drop or forward packets) than deliver to the user-space for further processing as in the experiment. Therefore, the approach is viable, performance-wise. For production usage, more throughout testing and assessing shall be conducted for each particular use-case.

Intervention and compatibility (R1B): Every application in this approach must include these 3 distinguish components: delivery, user-space program and kernel-space program.

- Both used delivery methods are provided by the Linux kernel with stable APIs and procedures. The kernel-space program is loaded into the kernel without permanently changing the platform and therefore is the compatibility requirement for this component fulfilled.
- The user-space program is an ordinary application that exchanges data with the kernel-space program through shared memory volume, which is also provided by the kernel, so there is no risk breaking compatibility here.
- The eBPF kernel program is only equipped with BPF libraries and system calls, which reduces the risk to minimal. The kernel module is exposed to the internal symbols and is prone to problems such as system calls become obsoleted, although such incidents rarely occur or can be simply updated to the new specification. For these reasons, we consider this component (and also the requirement) to be fulfilled.

Opportunity in production (R1C): The approach is relative simple and easy to implement with both methods having enough resources available to start with. The kernel program is the most challenging because due to its low-level tasks and the risks involved. In fact, some of these tasks are re-implementation of the network stack, which raises the question of reliability, efficiency and security of the implementation. In this case, eBPF (and XDP socket) has the advantage: most of these tasks had already been implemented and exposed as API for developers. Our kernel module program has 10 times the lines of code compared to the eBPF kernel program and the user-space program is also more complex. Nonetheless, the amount of work required for learning and writing the kernel program is relative small, taking into account the number of available example application and documentation for kernel's supported methods. We estimate that the approach is accordant with even small team of 1-3 developers with no special skill or knowledge required.

The eBPF method is only suitable for event-based processes while kernel module method is also commonly used for non-event-based, complex process like hardware driver, file system [p16]. Besides networking application, the approach can be applied for application that can have part of its logic living in the kernel, including high performance computing, monitoring or debugging. During the development, we also acknowledged that implementing the kernel-module program is much more challenging than eBPF kernel program. This can be explained by the fact that eBPF is a much more modern method, and also limits what the kernel program is allowed to do, which results in having much smaller interface than kernel-module program has. Performance-wise, the kernel-module method is slightly behind eBPF method, however that can be explained by the fact that eBPF has much better helper libraries, or because the test favored one method over the other.

8.4.2 Business requirements evaluation

Comparison to Linux performance: We covered the performance comparison with the **Performance improvement** requirement (R1A) entry, which clearly presents the difference in performance between Linux and the approach's eBPF and Kernel Module methods. Unfortunately, taking into account that collecting and processing data are too time-consuming and complex, the data regarding system's load and status during

the test couldn't be prepared and plotted for the final result. This leaves the complete comparison problem open for further work.

Performance comparison with other approaches: Due to time and resources constraints, we couldn't design and perform such sophisticated tests for direct comparison and so this research must be limited at investigating the ability of in-kernel processing approach and comparing to Linux network socket. We can only estimate the potential performance of the approach by learning from the other solutions that embedded 2 methods:

- Many researches showed that eBPF powered application can outperform the Linux network socket and in many cases matches DPDK capability, which is the golden rule for modern Linux networking real-time performance [p14] [p17].
- *PF_RING* and DPDK both also use kernel module programs to intercept and move the network packets to user-space for further processing, which makes them capable of handling 14M pps (million packets per second) [p14] [t26]. Our approach embeds real-time logic in the kernel space, therefore we can expect comparable performance from the approach.

License (R2C): Linux kernel is licensed under GPL2. This requires source code of any derivative work based on the kernel to be provided if the binary data is distributed. Many kernel's high-profile maintainers believe that Linux's kernel-module program and eBPF kernel program should fall under this term [m13] while the user-space program can be freely licensed. The kernel program can also be released as proprietary binary and although the kernel will allow these non-GPL-compatible modules and eBPF kernel program to be loaded, they will not have access to kernel symbols that are exported with *EXPORT_SYMBOL_GPL()*, which consisted of roughly 25% of the kernel's helpers [m12] [m7]. Listing 8.2 shows the gcc compiler complaining about the usage of GPL-only symbol in a non-GPL kernel module:

Listing 8.2: Our kernel module with GPL2 license removed is rejected during compiling due to usage of GPL-only symbol

```
1 make -C /lib/modules/5.8.0-63-generic/build M=/home/que/Desktop/mymodule modules
2 make[1]: Entering directory '/usr/src/linux-headers-5.8.0-63-generic'
3 CC [M] /home/que/Desktop/mymodule/kernel_module/intercept-module.o
4 MODPOST /home/que/Desktop/mymodule/Module.symvers
5 FATAL: modpost: GPL-incompatible module intercept-module.ko uses GPL-only symbol 'vfs_write'
```

This means, a proprietary application might keep the kernel program *closed-source*, however it will not be able to do much without the GPL-only symbol and therefore a production kernel program should be covered by GPL2 license. The developers now need to consider which business logic and algorithm can be disclosed in the kernel program, and keep the rest in the user-space program.

Technical support (R2D): Using this approach, the vendors can benefit from the stability of the Linux kernel and its ecosystem, although the open source nature also means 1. limited commercial support and 2. heavily relying on the community for advices and requests. This should not be considered as a drawback since the real-time solution is a niche market, therefore other solutions might not be able to react to customer's request as fast as the Linux's community can offer, especially when they are security related.

8.5 Summary of Requirements Evaluation

This table below summarizes the state of requirements:

Table 8.2: Evaluation of requirements

Research target	Requirement detail	Code	Evaluation
Functional requirements: Determining approach ability	Assess performance improvement	R1A	Fulfilled. The experiment shows improvement in delivery time (2-10 times), median, stability, ...
	Assert application's compatibility	R1B	Fulfilled.
	Viable in real-world application?	R1C	Fulfilled. eBPF and kernel module have been used for performant application related to real-time problem and are suitable even for small team.
Business requirements: Comparison performance	Compare to performance of Linux	R2A	Partly fulfilled. R1A entry covered the raw performance benchmark, however the performance penalty was not fully analyzed.
	Compare to other performance of other approaches	R2B	Not fulfilled. Because of time constraint, no other approaches have been successfully setup and tested. Our tests also were not designed to directly compared with existing published results from other researches.
	License	R2C	Fulfilled. The kernel program must be licensed under GPL2 to fully utilize the kernel internal APIs. However, important, proprietary logics can be implemented in user-space part and remain closed-source.
	Technical Support	R2D	Fulfilled. Also commercial support for Linux is very limited, the open-source community is very helpful and fast responsive. The technical documentations are also well maintained.

CHAPTER 9

SUMMARY AND FURTHER WORK

9.1 Summary

The thesis addresses the execution reliability issue of Linux kernel by using in-kernel processing approach. This allows application to move part of its logic into the kernel space for better resource assignment, earlier event handling and less overheads due to reducing communication and dependency, without making changes to the kernel and therefore preserving the kernel integrity and compatibility. Our experiments with the example applications proved the effectiveness of the approach with multiple times performance improvement over default Linux's solution, while introduced no permanent modification to the kernel. In the test cases, normal Linux socket server was used as the base-line to compare against applications built with the approach, with system in stressed and idle state scenarios. The obtained results suggested that while the Linux socket network performed well and stable overall, it was greatly out-performed by the approach in average latency value, worst-case value and ability to sustainably handle high workload.

There are 2 code-injecting methods that can be used to build such applications, namely eBPF and Kernel Module. eBPF is an easy to use, modern and safe method which can be used to solve many problems that couldn't be done outside of the kernel before, such as tracing. The kernel module method is less favorable due to its complexity and being hard to write secure and safe program [t13] compared to eBPF. Nonetheless, kernel module is a powerful method that has effectively unlimited freedom accessing the kernel resources, which can be a deciding factor over eBPF in certain use-cases.

The research promotes the existing in-kernel processing approach, demonstrates the enhancement it offers and advocates its usage in real-world application. Instead of using custom software and hardware, these methods can be utilized to build complex,

high-performance time-sensitive applications and help not only reducing production cost, development time and maintain effort, but also improving software compatibility by introducing the well-supported Linux kernel. This research aims to researchers, industry partners who are seeking a capable platform for a time-sensitive application, but not necessary be able to afford a custom software and hardware solution, or to replace an old system with a free and open Linux kernel. Although this approach might *not* be the safest option for applications that demand the utmost performance from the platform, it certainly is an elegant and long-term solution that offers comparable performance while remains simple and easy to maintain. For production system, we can recommend eBPF method for almost all use-cases. The kernel module method should be reminded for tasks that require beyond the capability of eBPF, such as non-event based task.

9.2 Further Work

Despite of the amount of time and effort spent on, we consider this research as still uncompleted due to the limitation in time and knowledge. The research can be extended and deepen in many of the existing threads: optimization for example application, more meaningful test scenarios, more data metrics to collect. The test applications were developed within a month during the research, using help from available resources such as example programs, books, blog posts ... Therefore, without proper professional supervisor and practices, the produced application is expected to be inefficient and might negatively affect the test results as well as conceal the actual potential capability. There are also features that were not implemented despite of having been planed such as *actual packet processing and client answering*, *real support for varied packet size* or *egress path support* for real-time sending packets. Depending on the use-case, more metrics would be interested or required: CPU load, buffer stress, context switching, ... We are also aware that the test scenarios could be improved, i.e introducing endurance test, more stressing-conditions (concurrent processes, variant networking condition, stress with CPU-bound, IO-bound, ...), which would require more time and effort to develop and setup. Most importantly, there are requirements that have not been satisfied, namely fully comparison with Linux and other approaches.

Through the research, we observed the difference between working with eBPF and Kernel Module: eBPF application can be implemented without putting in too much effort, however developers have only access to *bpf* libraries and toolings. Kernel module method on the other hand, has low-level flexibility thanks to access to all available kernel's symbols, although this also involves risks and demands more responsibility. Unlike eBPF, kernel module doesn't have modern, sophisticated libraries and helpers, which would introduce unnecessary re-implementation and possible bad quality code. This could be eliminated by a thin-framework that implements most of the inevitable initiation and management procedures. Developing such a framework/library would significantly speed up the process, improve the quality of the program and help shortening the learning curve.

APPENDIX A

KERNEL MODULE LICENSE

Following is a part of Linus Torvalds's discussion about if Linux Kernel Module fell under GPL license. The full mailing list discussion can be found at *https://lore.kernel.org/lkml/Pine.LNX.4.10.10312100538320.3805-100000@master.linux-ide.org/T/*

> Re: Linux GPL and binary module exception clause?
> 2003-12-03 21:31 Linux GPL and binary module exception clause? Kendall Bennett
> 2003-12-03 21:47 ‘ Arjan van de Ven
> 2003-12-03 22:11 ‘ Richard B. Johnson
> From: Linus Torvalds @ 2003-12-04 0:00 UTC (permalink / raw)
> To: Kendall Bennett; +Cc: linux-kernel
>
> On Wed, 3 Dec 2003, Kendall Bennett wrote:
> >
> > I have heard many people reference the fact that the although the Linux
> > Kernel is under the GNU GPL license, that the code is licensed with an
> > exception clause that says binary loadable modules do not have to be
> > under the GPL.
>
> Nope. No such exception exists.
>
> There's a clarification that user-space programs that use the standard system call interfaces aren't considered derived works, but even that isn't an "exception" - it's just a statement of a border of what is clearly considered a "derived work". User programs are _clearly_ not derived works of the kernel, and as such whatever the kernel license is just doesn't matter.
>
> And in fact, when it comes to modules, the GPL issue is exactly the same. The kernel _is_ GPL. No ifs, buts and maybe's about it. As a result, anything that is a derived work has to be GPL'd. It's that simple.
>
> Now, the "derived work" issue in copyright law is the only thing that leads to any gray areas. There are areas that are not gray at all: user space

is clearly not a derived work, while kernel patches clearly _are_ derived works.

But one gray area in particular is something like a driver that was originally written for another operating system (ie clearly not a derived work of Linux in origin). At exactly what point does it become a derived work of the kernel (and thus fall under the GPL)?

THAT is a gray area, and _that_ is the area where I personally believe that some modules may be considered to not be derived works simply because they weren't designed for Linux and don't depend on any special Linux behaviour.

Basically:
- anything that was written with Linux in mind (whether it then _also_ works on other operating systems or not) is clearly partially a derived work.
- anything that has knowledge of and plays with fundamental internal Linux behaviour is clearly a derived work. If you need to muck around with core code, you're derived, no question about it.

Historically, there's been things like the original Andrew filesystem module: a standard filesystem that really wasn't written for Linux in the first place, and just implements a UNIX filesystem. Is that derived just because it got ported to Linux that had a reasonably similar VFS interface to what other UNIXes did? Personally, I didn't feel that I could make that judgment call. Maybe it was, maybe it wasn't, but it clearly is a gray area.

Personally, I think that case wasn't a derived work, and I was willing to tell the AFS guys so.

Does that mean that any kernel module is automatically not a derived work? HELL NO! It has nothing to do with modules per se, except that non-modules clearly are derived works (if they are so central to the kenrel that you can't load them as a module, they are clearly derived works just by virtue of being very intimate - and because the GPL expressly mentions linking).

So being a module is not a sign of not being a derived work. It's just one sign that _maybe_ it might have other arguments for why it isn't derived.

Linus

---

* Re: Linux GPL and binary module exception clause?
2003-12-04 0:00 ‘ Linus Torvalds
@ 2003-12-04 0:23 ‘ Linus Torvalds
2003-12-04 6:25 ‘ Karim Yaghmour
2003-12-04 19:24 ‘ viro
2003-12-04 0:29 ‘ Kendall Bennett
From: Linus Torvalds @ 2003-12-04 0:23 UTC (permalink / raw)
To: Kendall Bennett; +Cc: linux-kernel

On Wed, 3 Dec 2003, Linus Torvalds wrote:
>

> So being a module is not a sign of not being a derived work. It's just
> one sign that _maybe_ it might have other arguments for why it isn't
> derived.

Side note: historically, the Linux kernel module interfaces were really quite weak, and only exported a few tens of entry-points, and really mostly effectively only allowed character and block device drivers with standard interfaces, and loadable filesystems.

So historically, the fact that you could load a module using nothing but these standard interfaces tended to be a much stronger argument for not being very tightly coupled with the kernel.

That has changed, and the kernel module interfaces we have today are MUCH more extensive than they were back in '95 or so. These days modules are used for pretty much everything, including stuff that is very much "internal kernel" stuff and as a result the kind of historic "implied barrier" part of modules really has weakened, and as a result there is not avery strong argument for being an independent work from just the fact that you're a module.

Similarly, historically there was a much stronger argument for things like AFS and some of the binary drivers (long forgotten now) for having been developed totally independently of Linux: they literally were developed before Linux even existed, by people who had zero knowledge of Linux. That tends to strengthen the argument that they clearly aren't derived.

In contrast, these days it would be hard to argue that a new driver or filesystem was developed without any thought of Linux. I think the NVidia people can probably reasonably honestly say that the code they ported had _no_ Linux origin. But quite frankly, I'd be less inclined to believe that for some other projects out there..

Linus

APPENDIX B

NETWORK DATA FLOW IN THE LINUX KERNEL

The Linux's network stack is a very complex structure with lots of waiting, queuing and buffering involved. Many of the approaches listed in this document are specifically designed to address the delay and unpredictability of the Linux networking. This image was downloaded from the archived version of *wiki.linuxfoundation.org*. The image can be zoomed in for more details.

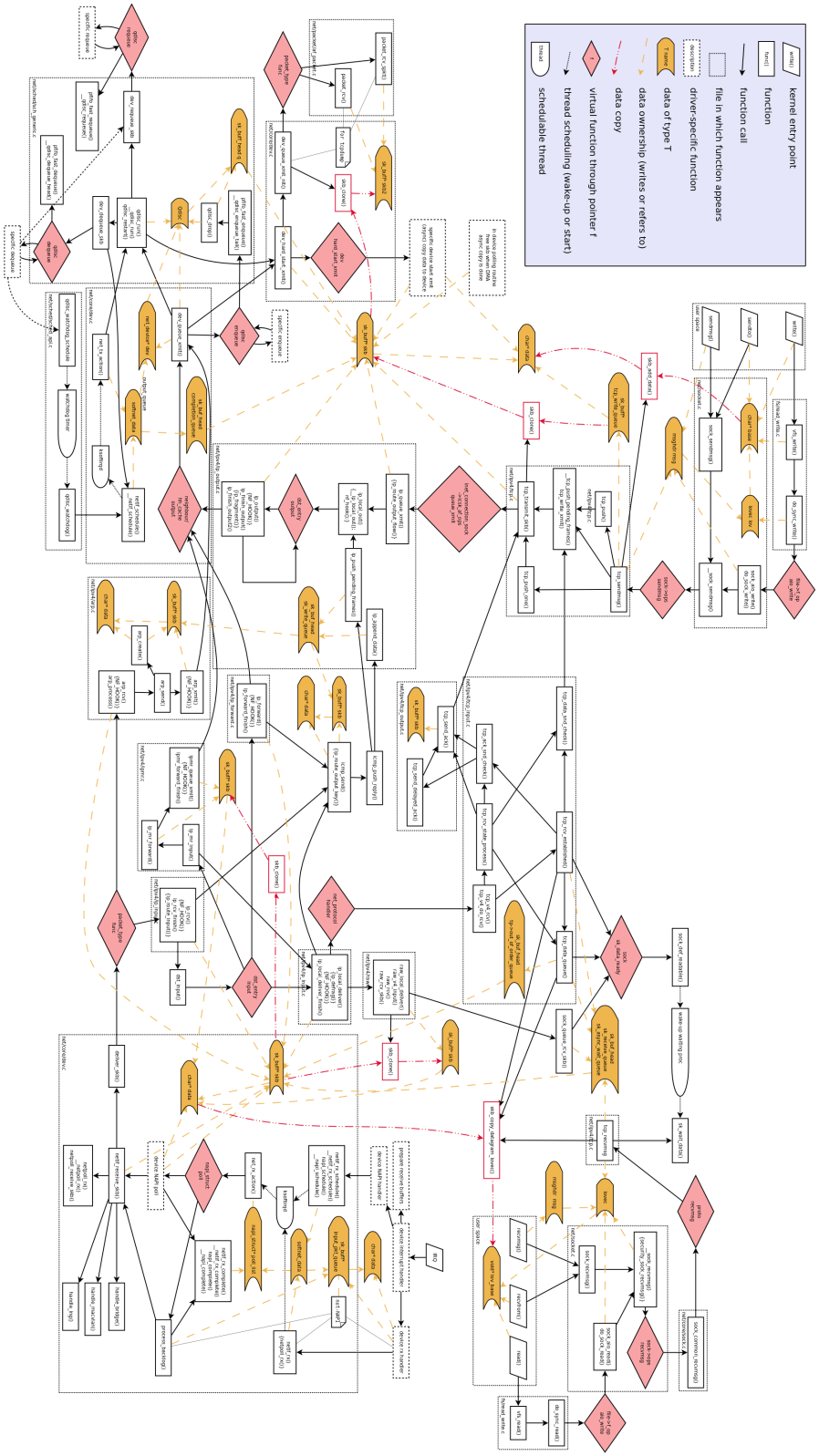


Figure B.1: Network data flow through kernel [m14].

ACRONYMS

API	Application Programming Interface	4
ASIC	Application-Specific Integrated Circuit	2
BPF	BSD Packet Filter	22
COTS	Commercial off-the-shelf	15
CPU	Central Processing Unit	11
DSP	Digital Signal Processor	2
eBPF	extended BPF	22
FPGA	Field-Programmable Gate Array	2
MMU	Memory Management Unit	12
NIC	Network Interface Card	31
OS	Operating System	iii
PC	Personal Computer	15
QoS	Quality of service	31
RTS	Real Time System	2
SDA	Software Defined Application	15
SDN	Software-defined networking	3
TCAs	Time Critical Applications	iii

BIBLIOGRAPHY

List of Author's Publications Covered in this Thesis

References to Scientific Publications

[p1] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. "A measurement-based analysis of the real-time performance of linux". In: *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*. ISSN: 1545-3421. Sept. 2002, pp. 133–142. DOI: 10.1109/RTTAS.2002.1137388 (cit. on p. 6).

[p2] G. G. Abraham Silberschatz and P. B. Galvin. *Operating System Concepts, Ninth Edition* (cit. on p. 12).

[p3] J. Altenberg. "Introduction to Realtime Linux". en. In: (), p. 42 (cit. on p. 17).

[p4] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha. "The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety". In: *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. ISSN: 1545-3421. Apr. 2009. DOI: 10.1109/RTAS.2009.20 (cit. on p. 2).

[p5] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly and Associates Inc, 2005, p. 8. ISBN: 0596005652 (cit. on p. 12).

[p6] D. P. Bovet, M. Cassetti, and A. Oram. *Understanding the Linux Kernel*. USA: O'Reilly, 2000. ISBN: 0596000022 (cit. on pp. 3, 4, 13–15).

[p7] E. W. Briao, D. Barcelos, F. Wronski, and F. R. Wagner. "Impact of task migration in NoC-based MPSoCs for soft real-time applications". In: *2007 IFIP International Conference on Very Large Scale Integration*. ISSN: 2324-8440. Oct. 2007. DOI: 10.1109/VLSISOC.2007.4402516 (cit. on p. 16).

[p8] D. Cerović, V. Del Piccolo, A. Amamou, K. Haddadou, and G. Pujolle. "Fast Packet Processing: A Survey". In: *IEEE Communications Surveys Tutorials* 20.4 (2018). Conference Name: IEEE Communications Surveys Tutorials, pp. 3645–3676. ISSN: 1553-877X. DOI: 10.1109/COMST.2018.2851072 (cit. on p. 19).

[p9] F. Cerqueira and B. B. Brandenburg. "A Comparison of Scheduling Latency in Linux, PREEMPT RT, and LITMUSRT". en. In: (cit. on p. 18).

[p10] P. Denning. "Virtual Memory". en. In: 1970 (cit. on pp. 12, 13).

[p11] N. Emamdoost, Q. Wu, K. Lu, and S. McCamant. "Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning". en. In: *Proceedings 2021 Network and Distributed System Security Symposium*. Virtual: Internet Society, 2021. ISBN: 978-1-891562-66-2. DOI: 10.14722/ndss.2021.24416. URL: https://www.ndss-symposium.org/wp-content/uploads/ndss2021_5B-4_24416_paper.pdf (visited on 06/04/2021) (cit. on p. 7).

[p12] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle. "A study of network stack latency for game servers". In: *2014 13th Annual Workshop on Network and Systems Support for Games*. 2014, pp. 1–6. DOI: 10.1109/NetGames.2014.7008960 (cit. on pp. 19, 31, 32).

[p13] C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, and V. M. Vilches. "Real-time Linux communications: an evaluation of the Linux communication stack for real-time robotic applications". en. In: *arXiv:1808.10821 [cs]* (Aug. 2018). arXiv: 1808.10821. URL: http://arxiv.org/abs/1808.10821 (visited on 05/31/2021) (cit. on pp. 16, 31).

[p14] 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". en. In: *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. Heraklion Greece: ACM, Dec. 2018, pp. 54–66. ISBN: 978-1-4503-6080-7. DOI: 10.1145/3281411.3281443. URL: https://dl.acm.org/doi/10.1145/3281411.3281443 (visited on 05/15/2021) (cit. on pp. 19, 53, 71).

[p15] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. 2021. URL: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-software-developers-manual-volume-1-basic-architecture.html (cit. on p. 13).

[p16] A. R. Jonathan Corbet and G. Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly Media, 2005. ISBN: 0-596-00590-3 (cit. on pp. 14, 43, 70).

[p17] M. Karlsson and B. Topel. "The Path to DPDK Speeds for AF XDP". en. In: (), p. 9 (cit. on pp. 53, 71).

[p18] M. Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. 1st. USA: No Starch Press, 2010, pp. 22–23. ISBN: 1593272200 (cit. on p. 12).

[p19] P. Koopman. "Embedded system design issues (the rest of the story)". In: *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*. ISSN: 1063-6404. Oct. 1996, pp. 310–317. DOI: 10.1109/ICCD.1996.563572 (cit. on p. 2).

[p20] H. Kopetz. "Real-Time Systems: Design Principles for Distributed Embedded Applications". en. In: Google-Books-ID: oJZsvEawlAMC. Springer Science & Business Media, Apr. 2011. ISBN: 978-1-4419-8237-7 (cit. on p. 2).

[p21] N. Litayem and S. Ben Saoud. "Impact of the Linux Real-time Enhancements on the System Performances for Multi-core Intel Architectures". en. In: *International Journal of Computer Applications* 17.3 (Mar. 2011), pp. 17–23. ISSN: 09758887. DOI: 10.5120/2202-2796. URL: http://www.ijcaonline.org/volume17/number3/pxc3872796.pdf (visited on 05/31/2021) (cit. on pp. 6, 16, 36).

[p22] A. Malinowski and H. Yu. "Comparison of Embedded System Design for Industrial Applications". In: vol. 7. Conference Name: IEEE Transactions on Industrial Informatics. May 2011. DOI: 10.1109/TII.2011.2124466 (cit. on p. 2).

[p23] S. McCanne and V. Jacobson. "The BSD Packet Filter: A New Architecture for User-Level Packet Capture". In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX'93. San Diego, California: USENIX Association, 1993, p. 2 (cit. on p. 7).

[p24] C. Poellabauer, A. Schwan, and R. West. "Lightweight kernel/user communication for real-time and multimedia applications". en. In: *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video - NOSSDAV '01*. Port Jefferson, New York, United States: ACM Press, 2001. ISBN: 978-1-58113-370-7. DOI: 10.1145/378344.378365. URL: http://portal.acm.org/citation.cfm?doid=378344.378365 (visited on 06/11/2021) (cit. on p. 6).

[p25] P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla. "On the Evaluation of the Impact of Shared Resources in Multithreaded COTS Processors in Time-Critical Environments". In: *ACM Trans. Archit. Code Optim.* 8.4 (Jan. 2012). ISSN: 1544-3566. DOI: 10.1145/2086696.2086713. URL: https://doi.org/10.1145/2086696.2086713 (cit. on p. 16).

[p26] F. Reghenzani, G. Massari, and W. Fornaciari. "The Real-Time Linux Kernel: A Survey on PREEMPT_RT". en. In: *ACM Computing Surveys* 52.1 (Feb. 2019). ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3297714. URL: https://dl.acm.org/doi/10.1145/3297714 (visited on 05/12/2021) (cit. on pp. 7, 15–17).

[p27] F. Reghenzani, G. Massari, and W. Fornaciari. "The Real-Time Linux Kernel: A Survey on PREEMPT_RT". en. In: *ACM Computing Surveys* 52.1 (Feb. 2019), p. 20. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3297714. URL: https://dl.acm.org/doi/10.1145/3297714 (visited on 05/18/2021) (cit. on p. 18).

[p28] P. Salzman, M. Burian, and O. Pomerantz. *The Linux Kernel Module Programming Guide*. 2009. ISBN: 9781441418869. URL: https://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html (cit. on pp. 41, 42).

[p29] V. Yodaiken. "The RTLinux Manifesto". In: (1999) (cit. on p. 17).

Technical References

[t1] *BPF Compiler Collection (BCC)*. Tech. rep. URL: https://github.com/iovisor/bcc (cit. on p. 22).

[t2] J. Brandeburg. *A way towards Lower Latency and Jitter*. en. Tech. rep., p. 28. URL: https://blog.linuxplumbersconf.org/2012/wp-content/uploads/2012/09/2012-lpc-Low-Latency-Sockets-slides-brandeburg.pdf (cit. on pp. 19, 32).

[t3] V. Bridgers. *Real Time Linux Scheduling Comparison*. en. Tech. rep. (cit. on p. 16).

[t4] *Building External Modules - kernel.org*. Tech. rep. URL: https://www.kernel.org/doc/html/latest/kbuild/modules.html (cit. on p. 44).

[t5] *CFS Scheduler*. Tech. rep. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html (cit. on p. 15).

[t6] *eBPF-based Networking, Observability, and Security*. Tech. rep. URL: https://cilium.io//how-netflix-uses-ebpf-flow-logs-at-scale-for-network-insight-e3ea997dca96 (cit. on p. 22).

[t7] *Frequently Asked Questions - RTwiki*. Tech. rep. URL: https://rt.wiki.kernel.org/index.php/Frequently_Asked_Questions (visited on 05/12/2021) (cit. on pp. 7, 16, 18).

[t8] J. Huang. *Effectively Measure and Reduce Kernel Latencies for Real-time Constraints*. en. Tech. rep. (cit. on p. 6).

[t9] *Improving Linux networking performance [LWN.net]*. Tech. rep. URL: https://lwn.net/Articles/629155/ (visited on 05/18/2021) (cit. on p. 31).

[t10] *Kernel module - ArchWiki*. Tech. rep. URL: https://wiki.archlinux.org/title/Kernel_module (visited on 06/08/2021) (cit. on p. 22).

[t11] *Kernel Module: Differences Between Versions Of Linux*. Tech. rep. URL: https://tldp.org/HOWTO/Module-HOWTO/linuxversions.html (cit. on p. 42).

[t12] *Kernel Modules Overview*. Tech. rep. URL: https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html (visited on 05/12/2021) (cit. on p. 7).

[t13] *Kernel Self-Protection*. Tech. rep. URL: https://www.kernel.org/doc/html/latest/security/self-protection.html (cit. on p. 75).

[t14] *Linux BPF header*. Tech. rep. URL: /include/uapi/linux/bpf.h#enum%20xdp_action (cit. on p. 56).

[t15] *Linux kernel licensing rules*. Tech. rep. URL: https://www.kernel.org/doc/html/latest/process/license-rules.html (cit. on pp. 21, 42).

[t16] *Linux kernel licensing rules*. Tech. rep. URL: https://www.kernel.org/doc/html/latest/process/license-rules.html#id1 (cit. on p. 42).

[t17] *Linux manual page: bpf*. Tech. rep. URL: https://man7.org/linux/man-pages/man2/bpf.2.html (cit. on pp. 35, 51, 52).

[t18] *Linux manual page: mmap*. Tech. rep. URL: https://man7.org/linux/man-pages/man2/mmap.2.html (cit. on pp. 35, 49).

[t19] *Linux process*. Tech. rep. URL: https://tldp.org/LDP/tlk/kernel/processes.html (cit. on p. 15).

[t20] *Memory Allocation Guide - kernel.org*. Tech. rep. URL: https://www.kernel.org/doc/html/latest/core-api/memory-allocation.html (visited on 06/08/2021) (cit. on p. 44).

[t21] *Message logging with printks - kernel.org*. Tech. rep. URL: https://www.kernel.org/doc/html/latest/core-api/printk-basics.html (cit. on p. 44).

[t22] *PREEMPT_RT patch versions*. Tech. rep. URL: https://wiki.linuxfoundation.org/realtime/preempt_rt_versions (cit. on p. 16).

[t23] *Real-Time group scheduling*. Tech. rep. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-rt-group.html (cit. on p. 15).

[t24] *Real-Time OS Kernels - Real Time Operating Systems and Middleware*. en-US. Tech. rep. URL: http://dit.unitn.it/~abeni/RTOS/kernels.pdf (cit. on p. 17).

[t25] Y. .-. Talk. *sched - overview of CPU scheduling*. Tech. rep. URL: https://man7.org/linux/man-pages/man7/sched.7.html (cit. on p. 12).

[t26] I. D. V. team. *DPDK Intel NIC Performance Report - Release 20.05 - Jun 10th 2020*. Tech. rep. URL: https://fast.dpdk.org/doc/perf/DPDK_20_05_Intel_NIC_performance_report.pdf (visited on 05/12/2021) (cit. on pp. 19, 39, 71).

[t27] *Understanding Linux Process States*. Tech. rep. URL: https://access.redhat.com/sites/default/files/attachments/processstates_20120831.pdf (cit. on p. 15).

[t28] *Virtual Memory Allocation And Paging - The GNU C library*. Tech. rep. URL: https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_3.html (visited on 05/12/2021) (cit. on p. 12).

[t29] *What Is A Kernel Module?* Tech. rep. URL: https://linux.die.net/lkmpg/x40.html (visited on 06/08/2021) (cit. on p. 22).

[t30] *XDP loader function by XDP tutorial*. Tech. rep. URL: https://github.com/xdp-project/xdp-tutorial/blob/master/basic02-prog-by-name/xdp_loader.c (cit. on p. 56).

Miscellaneous References

[m1] *A module for crashing the kernel*. Tech. rep. URL: https://lwn.net/Articles/371320/ (visited on 06/08/2021) (cit. on p. 22).

[m2] "A thorough introduction to eBPF [LWN.net]". In: URL: https://lwn.net/Articles/740157/ (cit. on pp. 7, 51).

[m3] "Aalto Talk with Linus Torvalds - Aalto University in Helsinki Finland". In: Linus Torvalds expressed his disappointment over Nvidia in front of the camera in an official talk. URL: https://youtu.be/MShbP3OpASA?t=2998 (cit. on p. 42).

[m4] W. Commons. "Privilege rings for the x86 available in protected mode". In: 2007. URL: https://en.wikipedia.org/wiki/Protection_ring#/media/File:Priv_rings.svg (cit. on p. 14).

[m5] *Deep Dive into Facebook's BPF edge firewall*. Tech. rep. URL: https://cilium.io/blog/2018/11/20/fb-bpf-firewall (cit. on p. 8).

[m6] *Detecting kernel memory leaks*. Tech. rep. 2006. URL: https://lwn.net/Articles/187979/ (visited on 05/13/2021) (cit. on p. 7).

[m7] *Document and clarify BPF licensing*. Tech. rep. URL: https://lwn.net/Articles/869396/ (cit. on p. 71).

[m8] *How Does Alibaba Cloud Build High-Performance Cloud-Native Pod Networks in Production Environments?* en. Tech. rep. URL: https://www.alibabacloud.com/blog/how-does-alibaba-cloud-build-high-performance-cloud-native-pod-networks-in-production-environments_596590 (visited on 05/13/2021) (cit. on p. 22).

[m9] "How Netflix uses eBPF flow logs at scale for network insight". In: URL: https://netflixtechblog.com/how-netflix-uses-ebpf-flow-logs-at-scale-for-network-insight-e3ea997dca96 (cit. on p. 22).

[m10] "How to receive a million packets per second". en. In: June 2015. URL: https://blog.cloudflare.com/how-to-receive-a-million-packets/ (visited on 05/18/2021) (cit. on p. 19).

[m11] "Kernel Developers Work To Block NVIDIA "GPL Condom" Effort Around New NetGPU Code". In: URL: https://lore.kernel.org/netdev/6376CA34-BC6F-45DE-9FFD-7E32664C7569@fb.com/T/#e556b6b762195afbbaf2782ba9be96ef9cf5e2e5c (cit. on p. 42).

[m12] *KRSI and proprietary BPF programs*. Tech. rep. URL: https://lwn.net/Articles/809841/ (cit. on p. 71).

[m13] *Linux GPL and binary module exception clause*. Tech. rep. URL: https://lore.kernel.org/lkml/Pine.LNX.4.10.10312100538320.3805-100000@master.linux-ide.org/T/ (cit. on p. 71).

[m14] *Network Data Flow through the Linux Kernel Diagram*. Tech. rep. URL: http://web.archive.org/web/20170905131225/https://wiki.linuxfoundation.org/images/1/1c/Network_data_flow_through_kernel.png (cit. on p. VI).

[m15] *Open-sourcing Katran, a scalable network load balancer*. en-US. Tech. rep. May 2018. URL: https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/ (visited on 05/13/2021) (cit. on pp. 8, 22).

[m16] "Some 5.16 kernel development statistics". In: URL: https://lwn.net/SubscriberLink/880699/63ad00ea7113bb26/ (cit. on p. 3).

INDEX

Q

R

S

T

U

V

X