Εθνικο Μετσοβιο Πολυτεχνειο
Τμημα Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Βελτιστοποίηση των λειτουργιών Εισόδου/Εξόδου στο λειτουργικό σύστημα linux με χρήση της τεχνολογίας eBPF

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Συμεών Ν. Ποργιώτης

Επιβλέπων: Γεώργιος Γκούμας
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2024

Εθνικο Μετσοβιο Πολυτεχνειο
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Βελτιστοποίηση των λειτουργιών Εισόδου/Εξόδου στο λειτουργικό σύστημα linux με χρήση της τεχνολογίας eBPF

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Συμεών Ν. Πoργιώτης

Επιβλέπων: Γεώργιος Γκούμας
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Φεβρουαριού 2024.

....	....	....
Ν. Κοζύρης	Δ. Πνευματικάτος	Γ. Γκούμας
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Αν.Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2024.

....

Συμεών Ν. Ποργιώτης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

© Συμεών Ν. Ποργιώτης, 2024, Εθνικό Μετσόβιο Πολυτεχνείο.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ' ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα. Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευτεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Με την εμφάνιση νέων τεχνολογιών αποθήκευσης δεδομένων υψηλών αποδόσεων, όπως οι ultra-low latency SSDs και οι NVMe συσκευές αποθήκευσης, είναι δυνατή η ολοκλήρωση διαδικασιών I/O σε χρόνους της τάξης των μικροδευτερολέπτων. Σε τέτοια υψηλή επίδοση, το προστιθέμενο κόστος από τη στοίβα αποθήκευσης του πυρήνα linux έχει γίνει σημαντικός παράγοντας στη καθυστέρηση ολοκλήρωσης των λειτουργιών αρχείων. Επιπλέον, καθώς οι συσκευές αποθήκευσης γίνονται ακόμη πιο γρήγορες, το σχετικό κόστος του πυρήνα αναμένεται να επιδεινωθεί. Το γεγονός αυτό αποτελεί κίνητρο για να επανεξετάσουμε τον τρόπο που ο πυρήνας διαχειρίζεται τις λειτουργίες αρχείων I/O και να βρούμε νέους τρόπους για τη μείωση του κόστους του.

Συγκεκριμένα, ο στόχος της παρούσας διπλωματικής εργασίας είναι να εξετάσει τη στοίβα αποθήκευσης του πυρήνα linux και να επικεντρωθεί σε ένα ιδιαίτερα κρίσιμο στοιχείο, την Page Cache, και τον τρόπο με τον οποίο επηρεάζει την απόδοση του συστήματος. Κατά βάση, εξετάσαμε τη προσπάθεια της Page Cache να βελτιώσει την απόδοση του συστήματος φέρνωντας δεδομένα προτού ζητηθούν μέσω του μηχανισμού Read-Ahead που έχει ως στόχο την αύξηση των ποσοστών ευστοχίας της. Επιπλέον, εξετάσαμε την αποτελεσματικότητά της σε διεργασίες που εμφανίζουν σειριακά πρότυπα πρόσβασης καθώς και τους περιορισμούς στην απόδοση της όταν εμφανίζονται διεργασίες με πιο πολύπλοκα πρότυπα πρόσβασης. Στα κεφάλαια που ακολουθούν, θα παρουσιάσουμε μια νέα επαναστατική τεχνολογία, το eBPF, καθώς και το πως μπορεί να χρησιμοποιηθεί για την ανάπτυξη ενός εργαλείου που θα αλλάξει τη συμπεριφορά του πυρήνα. Συγκεκριμένα, θα παρουσιάσουμε ένα νέο εργαλείο που έχουμε αναπτύξει, εκμεταλλευόμενοι την τεχνολογία αυτή, που επιτρέπει στους χρήστες να καθορίσουν ποια δεδομένα θα προστεθούν στην Page Cache, προσαρμόζοντας έτσι το μοτίβο πρόσβασης της στις ανάγκες των εφαρμογών τους. Τέλος, θα επισημάνουμε τα οφέλη της εκμετάλλευσης του νέου εργαλείου μας μέσω σύνθετων δοκιμών με τη βοήθεια του εργαλείου FIO για τις λειτουργίες αρχείων read() και mmap(), όπως επίσης και το πως μπορεί να χρησιμοποιηθεί για να βελτιώσει την επίδοση ενός προγράμματος σε πραγματικό περιβάλλον και συγκεκριμένα την μέθοδο snapshotting του Firecracker.

Λέξεις-Κλειδιά: πυρήνας του Linux, χώρος χρήστη, χώρος πυρήνα, Page Cache, eBPF, FIO, read, mmap, firecracker, snapshotting

Abstract

With the advent of new high performance storage technologies, such as ultra-low latency SSDs and NVMe storage devices, it is feasible to achieve I/O latencies at the scale of microseconds. At such high performance, the overhead introduced by the linux kernel storage stack has become a significant contributor to file operations latency. As storage devices become even faster, the kernel's relative overhead is going to worsen. This motivates us to reexamine the way the kernel handles file operations and find new ways to reduce its software overhead.

Specifically, the objective of this Diploma thesis is to examine the linux kernel storage stack and especially focus our attention on a crucial component, the Page Cache, and its impact on system's performance. Mainly, we delve into the effort made by the Page Cache to improve system's performance by pre-fetching data through the Read-Ahead mechanism in order to achieve high cache hit ratios. Additionally, we examine its effectiveness in sequential access patterns and its limitations in more complex access patterns. In the following Chapters, we will introduce a new revolutionary technology, the eBPF, and how it can be utilized to develop a tool that will alter the kernels behavior. In essence, we will present a new tool that we've developed, by leveraging this new technology, that will allow users to define what data will be added to the Page Cache and thereby how to customize the Page Cache fetching pattern to align it with the needs of their userspace applications. Finally, we will illustrate the benefits of exploiting our new tool through synthetic FIO benchmarks for read() and mmap() file operations and additionally we will provide insights on how it can benefit a real case scenario, specifically the firecracker's snapshotting method.

Keywords: Linux Kernel, Userspace, Kernel Space, Page Cache, eBPF, FIO, read, mmap, firecracker, snapshotting

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου υπό την επίβλεψη του Αναπληρωτή Καθηγητή Γεώργιου Γκούμα. Αρχικά λοιπόν, θα ήθελα να ευχαριστήσω τον κ.Γκούμα για την ευκαιρία που μου έδωσε να εκπονήσω τη διπλωματική μου εργασία στο συγκεκριμένο εργαστήριο.

Η εκπόνηση της διπλωματικής αυτής αποτελεί έμπνευση της Μεταδιδακτορικού Ερευνήτριας Χλόης Αλβέρτης. Θα ήθελα να ευχαριστήσω ιδιαιτέρως τη Χλόη όπως και τους Μεταδιδακτορικούς Ερευνητές Δημήτρη Σιακαβάρα, Στράτο Ψωμαδάκη και Χρήστο Κατσακιώρη που είχαν μια άριστη συνεργασία μαζί μου και με βοήθησαν στην ολοκλήρωση της διπλωματικής μου εργασίας. Τόσο η συνεχή τους καθοδήγηση και οι πολύτιμες γνώσεις που μου παρείχαν όσο και το ενδιαφέρον, η ενθάρρυνση, ο σεβασμός και η υπομονή που μου έδειξαν μετέτρεψαν τη διπλωματική μου εργασία σε μια πραγματικά σπάνια εμπειρία.

Επίσης, επιβάλλεται να ευχαριστήσω όλους μου τους φίλους με τους οποίους μοιράστηκα τα όμορφα φοιτητικά μου χρόνια. Κυρίως, θα ήθελα να πω ένα μεγάλο ευχαριστώ στο Λευτέρη και στη Κέλλυ, που ήταν οι πρώτοι με τους οποίους θα μοιραζόμουν την οποιαδήποτε επιτυχία ή αποτυχία συναντούσα είτε στα πλαίσια της σχολής είτε στη προσωπική μου ζωή.

Τέλος, δε γίνεται να μην ευχαριστήσω την οικογένεια μου και τους γονείς μου. Είναι οι άνθρωποι που όλα αυτά τα χρόνια με βοήθησαν να πετύχω τους στόχους μου και χωρίς την υποστήριξη, την κατανόηση, το ενδιαφέρον και την αμέτρητη αγάπη τους δε θα μπορούσα να έχω καταφέρει όλα όσα έχω καταφέρει σήμερα.

Συμεών Ποργιώτης

Contents

1 Introduction **12**
1.1 Current Solutions . . . 13
1.1.1 Polling . . . 13
1.1.2 Interrupt Driven I/O . . . 13
1.1.3 Scatter/Gather I/O . . . 13
1.1.4 Kernel Bypass . . . 13
1.2 Motivation: Page Cache . . . 14
1.2.1 Limitations of Kernel-Level Page Cache . . . 14
1.2.2 Limitations of User-Level Page Cache . . . 15
1.3 BPF : New Revolutionary Technology . . . 15

2 Kernel Stack **17**
2.1 System Call Layer . . . 18
2.2 Virtual Filesystem (VFS) . . . 19
2.3 Device Drivers . . . 21
2.3.1 Read Operation . . . 21

3 Page Cache **24**
3.1 How it Works . . . 24
3.2 Read-Ahead . . . 25
3.3 Synchronous Vs Asynchronous Read . . . 25
3.4 Read-Ahead Window . . . 26
3.5 Page Cache in Action . . . 27

4 eBPF **28**
4.1 Introduction . . . 28
4.1.1 Security . . . 29
4.1.2 Networking . . . 29
4.1.3 Tracing and Profiling . . . 29
4.1.4 Observability and Monitoring . . . 29
4.2 The evolution from BPF to eBPF . . . 29
4.2.1 The Evolution of eBPF to Production Systems . . . 30
4.3 How eBPF works . . . 31

4.3.1 eBPF Virtual Machine . . . 31
4.3.2 Hooks . . . 31
4.3.3 JIT Compiler . . . 32
4.3.4 Verifier . . . 32
4.3.5 Deployment of eBPF code . . . 34
4.4 High Performance of eBPF Programs . . . 34
4.5 eBPF Helpers . . . 34
4.6 eBPF Maps . . . 35
4.7 eBPF in Page Cache . . . 36
4.7.1 bpf_force_page2cache() . . . 37
4.7.2 offload_pages2cache() . . . 39
4.7.3 bpf_get_filename() . . . 40

5 Experimental evaluation **41**
5.1 Simulation Tool : fio - Flexible I/O tester . . . 41
5.2 Page Cache . . . 42
5.2.1 Latency . . . 42
5.2.2 read() . . . 42
5.2.3 mmap() . . . 44
5.3 Impact of "bpf_force_page2cache()" in Page Cache . . . 44
5.3.1 read() . . . 45
5.3.2 mmap() . . . 46
5.3.3 Results . . . 47
5.4 Use Case : Firecracker . . . 47
5.4.1 Serverless computing . . . 47
5.4.2 Cold Start Problem . . . 48
5.4.3 Snapshotting . . . 49
5.4.4 Deployment of eBPF . . . 50
5.4.5 Results . . . 50

6 Conclusions and Future Work **51**
6.1 Conclusions . . . 51
6.2 Future Work . . . 52

Chapter 1

Introduction

Storage performance is crucial in computer systems to ensure the continuous supply of data to a CPU without causing pipeline stalls and wasting CPU cycles. Traditionally, storage devices were much slower than CPUs and were considered the bottleneck of computer systems. Nowadays, the evolution of technology in storage devices has made things different. Today's ultra-low latency (ULL) SSDs and NVMe storage devices [1] can achieve sub-ten microseconds of I/O latency and have made kernel stack time comparable to hardware time thus creating the need to find new ways to decrease the latency of system operations.

For instance, when an application issues a read I/O request and a filemap fault occurs, then a new entry must be created in the Page Cache. After that, the entry must be filled with the requested data and several auxiliary data structures (e.g., bio) must be allocated and manipulated. The issue here is that these operations occur synchronously before an actual I/O command is dispatched to the storage device. With ultra-low latency SSDs, the time it takes to execute these operations is comparable to the actual I/O data transfer time, as depicted in Figure 1.1.

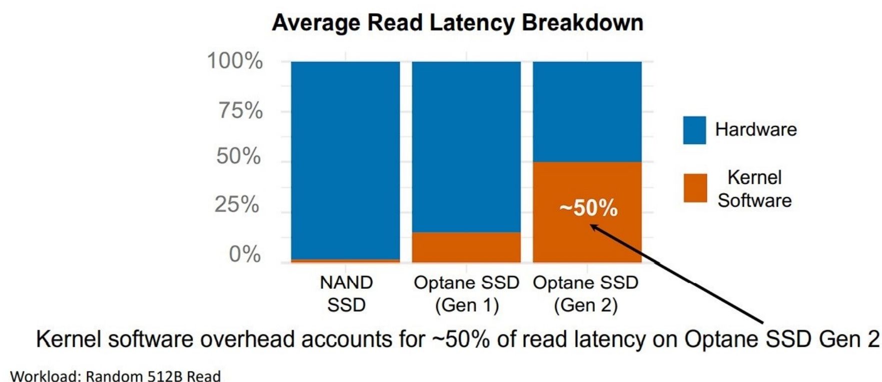

Figure 1.1: Kernel Software is becoming a bottleneck for system performance [17]

1.1 Current Solutions

Traditionally, the linux kernel is in charge of memory management and provides file abstraction to userspace processes. In this context, the first effort made to optimize the kernel I/O stack was to alleviate its overhead.

1.1.1 Polling

An effective approach to mitigate kernel I/O stack overhead is through the use of the polling mechanism. Polling a device [2] means repeatedly reading the device's status register to check if the I/O request has been completed. This mechanism is very useful to avoid time consuming context switches. As device drivers are integral parts of the linux kernel, employing a traditional "busy-wait" approach directly on the device would prevent any other tasks from running in the kernel during that time. Instead, polling device drivers utilize system timers to periodically check their status.

1.1.2 Interrupt Driven I/O

Polling by means of timers is at best approximate, since the kernel does not know when the device will be ready it could be wasting CPU cycles by constantly checking on it. A significantly more efficient method involves the use of interrupts. An Interrupt-driven I/O [2] device will create an interrupt when an event, such as an I/O completion, error or timeout has occurred. This mechanism consumes CPU cycles only when there is an event for the CPU to handle. However, it's important to note that the implementation of interrupt-driven I/O is notably more complex and challenging compared to the polling mechanism.

1.1.3 Scatter/Gather I/O

In cases where a userspace process needs to execute multiple read (or write) system calls to buffers that are scattered in memory, each system call must traverse the same kernel path, causing extra latency. To address this inefficiency, a high-speed primitive known as Scatter/Gather I/O [3] is employed to execute these scatter-gather operations in a single kernel call. As a result, this type of I/O takes its name from the fact that data is either scattered into or gathered from a given vector of buffers.

1.1.4 Kernel Bypass

These proposals are effective in reducing I/O stack overheads, and some of those are adopted by mainstream OSes (e.g., I/O stack for NVMe SSDs in linux). A different and more radical way to alleviate the I/O stack overhead would be to give userspace processes direct access to storage devices and completely bypass the

kernel. This approach can be achieved through libraries such as SPDK[4]. Such approach could be effective as it would leave the residual cost of posting a request to a storage device driver and the device's time to complete the request. On the other hand, it poses many challenges. Since processes don't have access to kernel features, each user has to construct his own userspace filesystem. A userspace filesystem means that there is no fine-grain isolation or sharing data between processes. In addition, there is no efficient way for userspace applications to receive interrupts on I/O completions and thus applications must directly poll on device completion queues to obtain high performance. Consequently, even though I/O is no longer the systems bottleneck, cores cannot be shared among processes and result in significant under-utilization of the CPUs. Furthermore when more than one polling thread shares the same core, the CPU contention between them, along side with the lack of synchronization, results in all polling threads to experience degraded tail latency and significantly lower overall throughput [5].

1.2 Motivation: Page Cache

Since data-intensive applications require a large amount of data transfer between a storage device and the systems main memory, operating systems employ Page Caches to efficiently manage slow data transfers. Page Cache is a crucial software component of the linux kernel and deeply affects a computer systems overall performance. Since the Page Cache is managed from the kernel it not only ensures robust data protection but also facilitates efficient data sharing across processes.

1.2.1 Limitations of Kernel-Level Page Cache

Kernel-Level Page Caches make an effort to exploit locality of reference [7] to reduce I/Os between disks and host memory. In order to exploit locality of reference, Page Cache makes use of the LRU (Least Recently Used) replacement policy [8] combined with Read-Ahead, a mechanism that makes an effort to reduce page faults by pre-fetching pages to Page Cache (we will delve into Read-Ahead in later chapters). Due to those characteristics, Page Cache generally achieves a high hit ratio if applications have moderate locality [9]. Unfortunately, when it comes to processes with more complicated I/O access patterns, like a graph application executing Pagerank algorithm [10], then this general-purpose design of the Page Cache fails to match specific I/O characteristics and shows sub-optimal performance. A possible reason for this poor performance could be either that a process might access a large range of I/O addresses repeatedly in a looping pattern and then the MRU (Most Recently Used) replacement policy [8] would outperform the commonly used LRU replacement policy or that a process could have a random access pattern to its I/O address space and then the standard Read-Ahead policy would prove ineffective.

Page Cache Hints

Userspace processes can provide hints to the linux kernel by pre-declaring an access pattern for a file operation via fadvise [11] and/or giving directions about the address range via madvise [12]. At the same time, processes retain the same advantages of the kernel-level cache management. However, this method can provide limited results as it is very hard to fine control the Page Cache just by injecting hints to the kernel and additionally, users must make a non-trivial effort to modify their applications in order to match their I/O access pattern with the provided hint to the linux kernel.

1.2.2 Limitations of User-Level Page Cache

Some data-intensive applications such as Jaydio [13] or RocksDB's Direct-IO [14] with complex I/O access patterns implement their own userspace Page Caches instead of relying on the Kernel-Level Page Cache and User-Level hints. On the one hand these applications show excellent cache hit ratios which greatly improve systems performance. On the other hand, they require developers to create the caching algorithm from scratch and can not take advantage of kernel's data protection and consistency or its sharing features thus making applications more vulnerable to data protection issues and requiring extra effort to enable data sharing. Another major drawback is that userspace Page Cache is actually located inside the kernel Page Cache so its performance depends on the applications dataset size. If the dataset exceeds the application's cache memory then kernel Page Cache makes use of the LRU replacement policy evicting pages with possibly useful data to the disk and therefore deteriorating applications performance. Moreover, if there are significant changes in either disk or the host memory system then the userspace cache would need to be re-implemented.

1.3 BPF : New Revolutionary Technology

The kernel-level Page Cache faces challenges in delivering optimal performance, primarily because it lacks the capability to adapt to application-specific I/O patterns. Although application-level hints offer some mitigation, their effectiveness is limited compared to an ideal solution. While User-Level custom Page Caches demonstrate efficiency, they are unable to harness the kernel's infrastructure and suffer a performance drop due to kernel intervention. To address these challenges, we rely on BPF (Berkeley Packet Filter [15]) which lets applications offload simple functions to the linux kernel. Specifically, in order to overcome the issues mentioned above, we leverage BPF technology to create an interface that enables users to determine which pages of a specific process will be loaded into the Page Cache before the process triggers a Page Cache miss.

In contrast to kernel bypass and User-Level Page Caches, BPF is an OS-supported mechanism that ensures isolation, avoids low utilization due to busy-waiting, and allows a large number of threads or processes to share the same core, leading to better overall utilization. Linux's framework for BPF is called eBPF (extended BPF [16]). eBPF is commonly employed for various purposes, including filtering packets (e.g., TCPdump), load balancing, packet forwarding, tracing, packet steering, network scheduling, and network security checks. In the final Chapters of this Diploma thesis, we will examine in detail the eBPF technology and how we can exploit it to to develop the tool that will achieve our goal.

Chapter 2

Kernel Stack

In this chapter we will delve into the read file operation. Accessing a file located on disk for a read() operation is a complex activity that involves many different layers of the linux kernel, as well as handling Block Devices and using the Page Cache. This complexity arises from a fundamental principle of the linux kernel which strictly prohibits applications in userspace from directly accessing hardware devices. Instead, linux offers processes running in userspace a set of interfaces to interact with hardware devices such as the CPU, disks, etc.

When a userspace application tries to read the contents of a file then that could either be a short trip to the L1 CPU cache [18] or a long trip to a storage device. Naturally, a short trip to a CPU cache provides minimum latency and is the fundamental reason for which hardware caches were introduced – to alleviate the performance mismatch between the CPU and RAM. Specifically, caches are based on the locality principle which states that when some data is requested then the data next to it have a high probability of being used in the near future. This results in the cache controller selecting whole cache lines to transfer instead of just the requested data. For example, when a cache hit occurs during a read file operation the controller selects the data from the cache line and transfers it into a CPU register. On the other hand, when a cache miss occurs, then the cache line is written to memory, if necessary, and the correct line is fetched from the RAM into a cache entry. Again if the data is present in the RAM then they get transferred to the cache and if not they are fetched from a storage device. Unfortunately, when new data are requested from a process, for example because the process just started running, then they are not available in any of the CPU caches or the systems RAM and thus the kernel must fetch them from a storage device.

In the following section, we will examine the linux kernel software layers involved in fetching data from a storage device for a userspace process.

2.1 System Call Layer

A process needs to interact with the linux kernel [19] so that the kernel can perform a set of operations on its behalf. For example, if a userspace process needs to do some sort of I/O, like open, read, write, etc, or modify its address space, with mmap or sbrk then it has to trigger the kernel. What prevents a userspace application from performing these type of operations on its own is its privilege level.

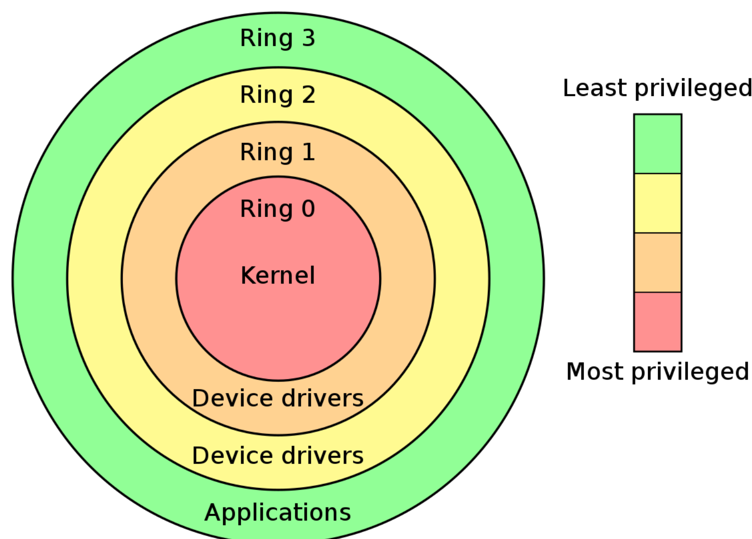

Figure 2.1: Userspace applications belong to Ring 3 and have the least privileges in a computer system

The architecture of modern computer systems obliges that there is a secure model. The picture above demonstrates how this is achieved through the “protection rings” model, which specifies multiple privilege levels under which software may be executed. For example, a userspace process belongs to the outer ring. That means that it is limited to its own address space, so that it can’t access or modify other running processes or the linux kernel and is prevented from directly accessing hardware devices. In other words, the linux kernel considers all running in userspace processes to be malicious and trusts only itself to execute commands.

In order for an application to gain access to a hardware device, for example to read the contents of a file located in a storage device, system calls are made available from the linux kernel to provide well-defined and safe implementations of such file operations. The linux kernel is the most privileged component of a computer system and has access to everywhere. A system call is essentially an entry point for a userspace process into the linux kernel. Usually, a system call is not directly invoked but instead a corresponding C library wrapper function is

called to perform the steps required to invoke a system call [20]. In most cases these steps are :

- copying arguments and the unique system call number to the registers where the kernel expects them.
- trapping to kernel mode, at which point the kernel does the real work of the system call.
- setting errno if the system call returns an error number when the kernel returns the CPU to user mode.

A system call is initiated via a software interrupt. A software interrupt automatically puts the CPU into some elevated privilege level and then passes control to the kernel. After that, the kernel determines whether the calling process should be granted the requested service by checking the accuracy of the request at the System Call Layer before attempting to satisfy it. This involves checking all system call parameters and if any of them contains an address, then to check whether it is inside the process address space. If the service is granted then the kernel executes a specific set of instructions over which the calling process has no direct control until the completion of the request and the return of the CPU to userspace.

Last but not least, it is important to mention that the System Call Layer has been kept stable through a policy of not introducing changes to it. This stability guarantees the portability of applications source code.

2.2 Virtual Filesystem (VFS)

The service routine of a system call, for example a `read()` system call, activates a suitable Virtual Filesystem function, in the case of a `read()` system call it passes to it a file descriptor and an offset inside the file. The Virtual Filesystem (VFS) is the second layer of the kernel stack and handles all system calls related to a standard Unix filesystem. The VFS can be seen as the upper layer of the block device handling architecture.

Due to the emergence of different kinds of filesystems such as ext2, ext3, NTFS and others, the VFS was developed by the linux kernel to allow userspace processes to access all filesystems in a uniform way. For example, VFS is used to access local and network storage devices transparently without the running userspace process noticing the difference.

Essentially, VFS consists of a wide range of information about every supported filesystems operations and is stored as a part of the linux kernel [21]. For instance when a read, write, or some other system call is executed, the kernel with the help of the VFS substitutes the system call with the actual function that is supported from the native linux filesystem, the NTFS filesystem, or whichever other filesystem the file is on.

In the case of a `read()` system call, the first step performed from the VFS is to determine whether the requested data are already available and if not how to perform the `read()` operation. Sometimes there is no need to access the data on disk because the kernel keeps in RAM the data most recently read from or written to a block device. If a `read()` operation must be performed then the VFS is responsible to translate it into a call to the corresponding filesystems function. Every file in a filesystem is represented by a file data structure which contains a field that has pointers to functions specific to files, such as the read function. The VFS finds the pointer to this function and invokes it. Once the requested data are fetched from the storage device and become available in the system's RAM, the kernel can return them.

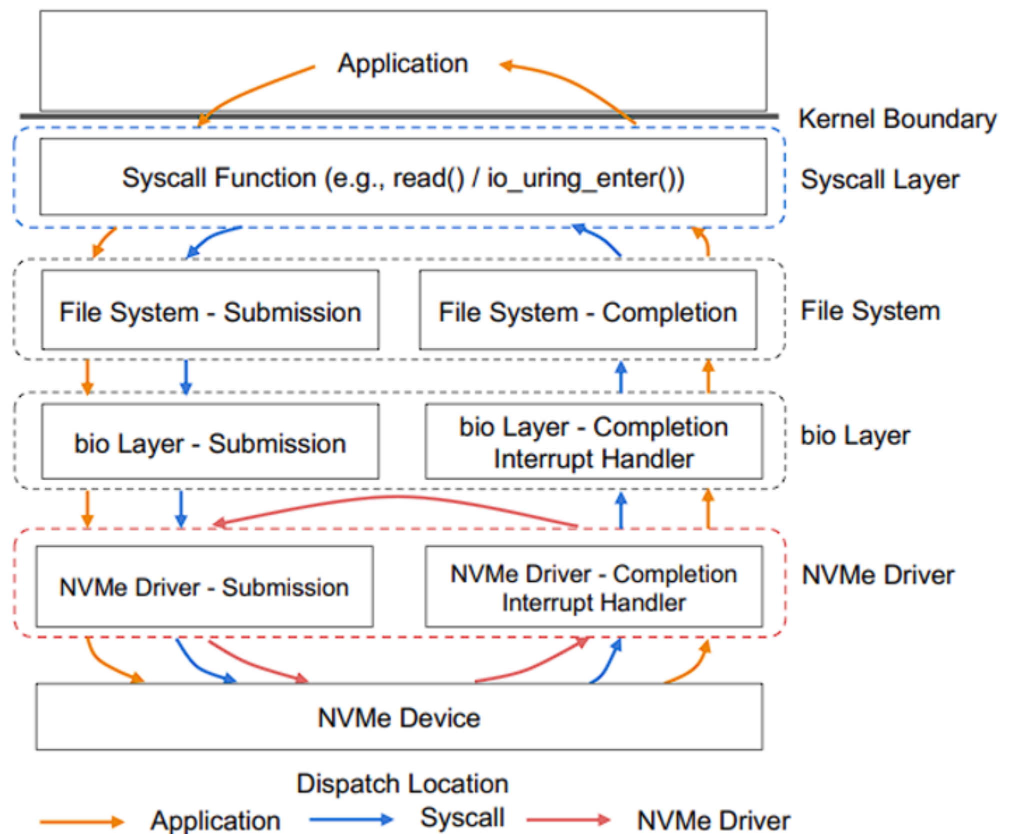

Figure 2.2: The Linux Kernel Stack

2.3 Device Drivers

A Device Driver is a software component of the linux kernel that operates or controls a particular type of device that is attached to a computer system. Essentially, a Device Driver is the software interface that enables the linux kernel to access a hardware device and respond to the programming interfaces defined by the canonical set of VFS functions (open, read, lseek, ioctl, and so forth) without requiring detailed knowledge of the specific hardware in use.

Block Device Driver

A special category of Device Drivers are the Block Device Drivers [21] which are the lowest components of the linux block subsystem. For a Block Device Driver the key aspect is the disparity between the time taken by the CPU and the buses, to read (or write) data and the speed of the disk. Block devices have very high average access time and each operation requires several milliseconds to complete. Since accesses to these devices are usually very slow, the kernel provides sophisticated components such as the Page Cache and the block I/O subsystem to handle them.

Disk

A disk is a logical Block Device such as either a hardware disk (HDD) or a virtual device built upon several physical disk partitions or even a storage area living in some dedicated pages of RAM. In any case, different types of disks are handled in the same way by the upper kernel components thanks to the services of the Block Device Drivers. Essentially, the linux kernel sees a disk as a split in blocks, each block corresponds to the minimal disk storage unit defined from a filesystem, and each block is assigned a Logical Block Number (LBA) which is its identifier.

2.3.1 Read Operation

In order for the kernel to read data from a Block Device, such as a Hard Disk, it must first determine their physical location on the device. Initially, it must determine what is the block size of the filesystem that the requested file belongs to and then compute which blocks of the file are requested. A file is divided into blocks and each block is assigned a File Block Number (FBA) as its identifier. This identification is crucial as the filesystem maintains a mapping between file block numbers and their corresponding Logical Block Numbers (LBA) on the storage device. This mapping enables the filesystem to track the storage location of each block of a file. Once the kernel has determined the requested FBAs it can establish the physical location of these blocks inside the disk by checking their corresponding

LBAs. It is important to notice that due to the mapping of FBAs to LBAs a file may be stored in nonadjacent blocks on a disk.

Since the physical position of the requested data has been located the kernel can now perform a read operation on the Block Device. When a kernel component wishes to read (or write) some disk data thanks to the help of the Block Device Drivers it simply must create a Block Device request. This request is submitted to the generic block layer which is the software component responsible for handling all requests to block devices and glues together all the upper and lower components. Since the requested data is not necessarily adjacent on disk, the generic block layer might need to start several I/O operations. Each I/O operation is represented by a "block I/O" structure or in short, "bio", which collects all information needed by the lower components to satisfy the request. The generic block layer hides the peculiarities of each hardware block device, thus offering an abstract view of the block devices. Below the generic block layer, the "I/O scheduler" sorts the pending I/O data transfer requests according to predefined kernel policies. The purpose of the scheduler is to group requests of data that lie near each other on the hardware device. Finally, the block device driver takes care of the actual data transfer by sending suitable commands to the hardware interfaces of the disk controllers.

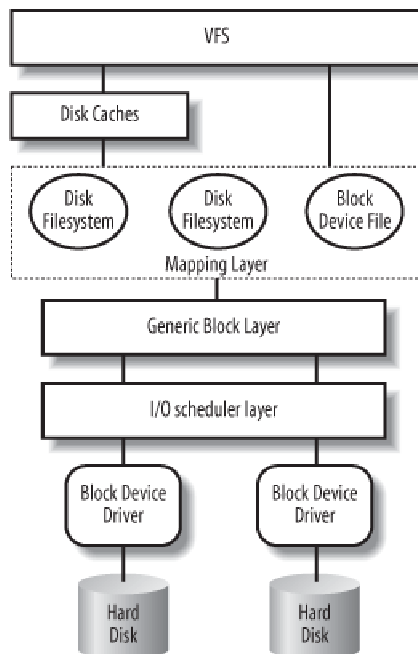

Figure 2.3: Abstract Image of a block device operation [21]

In particular, block device drivers are interrupt-driven which means that the generic block layer invokes the I/O scheduler to create a new block device request or to enlarge an already existing one and then terminates. The Block Device Driver, which is activated at a later time, invokes the strategy routine to select

a pending request and satisfy it by issuing suitable commands to the disk controller. When the I/O operation terminates, the disk controller raises an interrupt and the corresponding handler invokes the strategy routine again, if necessary, to process another pending request. It is important to notice that when a new block data transfer is requested, the kernel checks whether it can be satisfied by slightly enlarging a previous request that is still waiting. Due to the fact that disks tend to be accessed sequentially, this simple mechanism greatly improves systems performance.

Chapter 3

Page Cache

Up to this point, we have examined how the linux kernel fetches data from a storage device. In this Chapter we delve into one of the most crucial components of a computer system: the Page Cache. The Page Cache is a specific type of disk cache responsible of keeping in RAM data that is normally stored on a disk, such as the contents of a regular file. The importance of the Page Cache to the performance of a computer system lies in two key aspects :

1. further accesses to the same data can be satisfied quickly without accessing the disk again as the data is "cached" in the Page Cache.
2. Page Cache has developed a mechanism called Read-Ahead to improve systems performance by pre-fetching data.

3.1 How it Works

First of all, Page Cache works on whole pages of data. We use the term "page" to refer both to a set of linear addresses and to the data contained in this group of addresses. In particular, for a page size of 4 KB, we denote the linear address interval ranging between 0 and 4,095 as page 0, the linear address interval ranging between 4,096 and 8,191 as page 1, and so forth. Each memory region therefore consists of a set of pages that have consecutive page numbers.

Practically, file operations such as read and write rely on the Page Cache to get satisfied. When a process tries to read (or write) the data of a file then it actually tries to read (or write) the set of pages that contain the file's data. If the requested pages are available then the Page Cache simply returns them. If not, then the kernel has to take a series of steps to fulfill the request. The first step is for the kernel to allocate new page frame(s) in the systems RAM which later on will be filled with new page(s). Reading a page of data from a file is just a matter of finding what blocks on disk contain the requested data. It is important to mention that a page does not necessarily contain physically adjacent disk blocks. For that

purpose, the VFS calls the filesystems generic function which is used to implement the read method for block device files and for regular files. Once this is completed, the kernel fills the pages by submitting the proper I/O operations to the generic block layer. Once the pages are filled with the requested data the kernel copies them into the process address space. If there is enough free memory then the page is kept in the cache for an indefinite period of time and can be reused by other processes without the need to access the disk. If there is not enough memory, then the linux kernel must perform a method called swap() to release pages from Page Cache and to create space for new pages.

3.2 Read-Ahead

Predicting the future is hard a problem, but the kernel has to make an effort to do so through the Read-Ahead mechanism. Read-Ahead is a mechanism developed by the linux kernel which consists of reading several adjacent pages of data before they are actually requested. Read-Ahead only ever attempts to read pages that are not yet in the Page Cache and if a page is present but not up-to-date, then it will not try to read it.

In most cases, Read-Ahead significantly enhances disk performance. Specifically, there are situations where it is not too hard to predict what will happen next, namely, when a process sequentially accesses a file. For example, consider a situation in which a process reads a file linearly from position A to position B. It therefore makes sense to Read-Ahead from position B until position C so that when requests for pages between B and C are issued from the process, the data will already be available in the Page Cache.

3.3 Synchronous Vs Asynchronous Read

In the previous example, a process issued a read request from position A to position B of a file. Let's say that the file's data from position A to B is contained in 8 pages. Since the process hasn't tried to read the file before, those pages will not be available in the Page Cache. Thus, the kernel will perform the necessary steps to fetch these 8 pages from the disk. This type of read request is called a Synchronous read (Sync read) because the process has to wait for the kernel to fetch the corresponding pages to continue its execution. Moreover, the kernel must set a marker called PG_Readahead, for example on the sixth page, to signal the beginning of an Asynchronous read (Async read).

Once the pages are available in the Page Cache the process sequentially reads them. When the sixth page is accessed the kernel notices that the page was equipped with the PG_Readahead marker. This triggers an Asynchronous read operation that fetches a number of pages in the background. For example, the kernel fetches 16 more pages to the Page Cache, which in our example contain

the data from position B to C of the file. This type of read request is called an Asynchronous read (Async read) because pages are fetched before they are actually required. Since two more pages are left in the Page Cache from the initial read from position A to B, there is no need to hurry. Meanwhile, the I/O performed in the background will ensure that the pages are present when the process makes further progress in the file. As a result, when the process tries to read the contents of the file from position B to C the data will already be contained in the Page Cache. If the kernel would not adopt this scheme, then Read-Ahead could only start after a process has experienced a page fault. Thus the required page would be brought into the Page Cache Synchronously and this would introduce delays which reduce system's performance. This scheme is now repeated further. The Asynchronous read marks a page with the PG_Readahead flag, and the kernel will start a new Asynchronous read when the process has accessed the marked page, and so on.

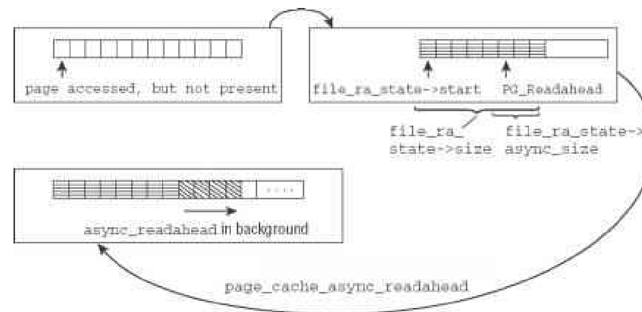

Figure 3.1: Overview of the Read-Ahead mechanism [22]

3.4 Read-Ahead Window

Adding pages to the Page Cache is simple task from a technical point of view. However, the Page Cache has a limited space that should not be occupied by pages that will not be accessed. More importantly the key-point of Read-Ahead is to have activated the low-level I/O device driver at the proper time so that the future pages will have been transferred when the process needs them. In the end, the challenge lies in predicting the optimal number of pages that the kernel should pre-fetch. For that reason, the Page Cache requires a sophisticated algorithm that will increase the number of pre-fetched pages when noticing a sequential access and decrease them or even disable Read-Ahead otherwise.

For this purpose, while accessing a given file, the Read-Ahead algorithm makes use of two sets of pages, each of which correspond to a contiguous portion of the file. These two sets are called the current window and the Read-Ahead window. The current window consists of pages fetched from a Sync read and contains both the last pages sequentially accessed by the process and possibly some of the pages

that have been read in advance by the kernel but have not yet been requested by the process. The Read-Ahead window consist of pages fetched from an Async read which follow the ones in the current window. No page in the Read-Ahead window has yet been requested by the process, but the kernel assumes that sooner or later the process will request them.

When a process accesses a file for the first time, the kernel creates a new current window starting from the first requested page. All current windows have a length that is a power of two and their initial size mainly depends on the number of requested pages. As soon as the kernel recognizes that the process has performed a sequential access in the current window it creates the Read-Ahead window. The length of the Read-Ahead window depends on the length of the current window and it is either two or four times larger from it. If the process continues to access the file in a sequential way, eventually the process will request pages that belong to the Read-Ahead window. Thus, the Read-Ahead window will become the new current window and a new one will be created. As a result, Read-Ahead is aggressively enhanced if the process reads the file sequentially. It is worthwhile mentioning, that when a process accesses a file for the first time, the kernel automatically presumes that it will make a sequential access on it.

3.5 Page Cache in Action

The core idea in Read-Ahead is to read more pages than those requested. If successful and the extra pages are accessed then, that would justify taking a risk to read even more data that hasn't been requested. However, if the kernel recognizes that the file access is not sequential, the current and Read-Ahead windows are cleared (emptied) and the Read-Ahead is temporarily disabled. When the kernel notices that a file has started to get accessed sequentially again then, it restarts the Read-Ahead from scratch.

The linux kernel should be able to provide efficient performance for processes that have sequential and/or random access patterns over a file. Unfortunately, for those with random access patterns Read-Ahead is of no use and it could even deteriorate the Page Cache performance because it tends to waste space with useless information. We will examine this situation in Chapter 5.

Chapter 4

eBPF

4.1 Introduction

eBPF [23], acronym for extended Berkeley Packet Filter, is a revolutionary kernel technology that allows developers to write custom code that can be loaded into the kernel dynamically, changing the way the kernel behaves. eBPF provides a secure and efficient method to enhance kernel capabilities during runtime without requiring changes to kernel source code or loading kernel modules.

Safety is ensured through an in-kernel verifier that conducts static code analysis, rejecting programs that could potentially cause issues to the kernel. Examples of programs that are automatically rejected by the verifier include those that might trap the kernel in infinite loops (e.g., for/while loops without exit conditions) and programs dereferencing pointers without safety checks.

Loaded programs which passed the verifier are interpreted through an in-kernel JIT compiler, utilized to achieve native execution performance. The execution model is event-driven, allowing programs to be attached to various hook points in the kernel and executed upon triggering of an event.

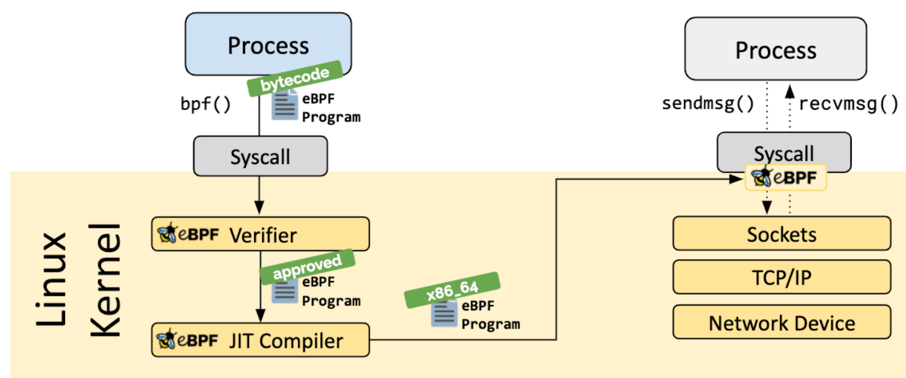

Figure 4.1: Abstract image of an eBPF program

Below we present some of the most common use cases for the eBPF technology.

4.1.1 Security

Typically, entirely independent systems have handled different aspects of security. For example, we would need independent systems for system call filtering, process context tracing, and network-level filtering. In contrast, eBPF facilitates the combination of control and visibility over all aspects by extending the basic capabilities of seeing and interpreting all system calls and providing packet and socket-level views of all networking operations. This allows the development of security systems that operate with more context and an improved level of control.

4.1.2 Networking

eBPF allows developers to install faster, more tailored packet processing features, load balancing processes, application profiling scripts and network monitoring practices. Open-source platforms, like Cilium, leverage eBPF to provide secure, scalable networking for Kubernetes clusters and workloads, and other containerized microservices. Furthermore, by leveraging kernel-level package forwarding logic, eBPF can streamline routing processes and enable faster overall network response [25].

4.1.3 Tracing and Profiling

The ability to attach eBPF programs to kprobes, uprobes and tracepoints enables the user to profile the runtime behavior of both userspace and kernel space processes. This gives unique and powerful insights to troubleshoot system performance issues. Also, advanced statistical data structures allow users to extract useful data in an effective way, without needing to export huge amounts of sampling data that is typical for similar systems.

4.1.4 Observability and Monitoring

eBPF lets developers instrument the kernel and userspace applications to collect detailed performance data and metrics without significantly impacting the system's performance. These capabilities enable real-time monitoring and observability.

4.2 The evolution from BPF to eBPF

What we call "eBPF" today has its roots in the BSD Packet Filter, first described in a paper [26] written by Lawrence Berkeley National Laboratory's Steven McCanne and Van Jacobson. This paper discusses a pseudomachine that can run

filters, which are programs written to determine whether to accept or reject a network packet. These programs were written in the BPF instruction set, a general-purpose set of 32-bit instructions that closely resembles assembly language. The heart of what eBPF enables is that the author of the filter can write their own custom programs to be executed within the kernel without the need to change the kernel.

BPF came to stand for “Berkeley Packet Filter” and it was first introduced to linux in kernel version 2.1.75 [27], where it was used in the tcpdump utility as an efficient way to capture the packets to be traced out. Fast-forward to when seccomp-bpf was introduced in version 3.5 of the kernel. This enabled the use of BPF programs to make decisions about whether to allow or deny userspace applications from making system calls. This was the first step in evolving BPF from the narrow scope of packet filtering to the general-purpose platform it is today. From this point on, the words packet filter in the name started to make less sense as BPF demonstrated capabilities beyond that!

BPF evolved to what we call “extended BPF” or “eBPF” in kernel version 3.18, while the former version became “classic BPF” or “cBPF”. At that time, “eBPF” was completely reshaped, with the addition of new functionalities and performance improvements. New features such as maps and tail calls appeared and the JIT compiler was rewritten in order to provide a new language that would be even closer to native machine language than cBPFs was.

4.2.1 The Evolution of eBPF to Production Systems

A feature called kprobes [28] (kernel probes) had existed in the linux kernel for years, allowing for traps to be set on almost any instruction in the kernel code. Developers could write kernel modules that attached functions to kprobes for debugging or performance measurement purposes.

The ability to attach eBPF programs to kprobes was the starting point for a revolution in the way tracing is done across linux systems. At the same time, hooks started to be added within the kernel’s networking stack, allowing eBPF programs to take care of more aspects of networking functionality.

Due to eBPFs popularity it became a separate subsystem within the linux kernel. At the same time, BPF Type Format (BTF) was introduced, which made eBPF programs much more portable. The final step in eBPFs evolution was the introduction of LSM BPF, allowing eBPF programs to be attached to the Linux Security Module (LSM) kernel interface. This indicated that a third major use case for eBPF had been identified. It became clear that eBPF is a great platform for security tooling, in addition to networking and observability.

4.3 How eBPF works

4.3.1 eBPF Virtual Machine

An eBPF program is a set of eBPF bytecode instructions that run in an in-kernel execution environment called the eBPF virtual machine [30]. It's possible to write eBPF code directly in this bytecode form, much as it's possible to program in assembly language. However, this would prove to be rather difficult than useful and instead developers first write their programs in a higher-level programming language, such as C, that is easy to deal with. Once the developer writes his code, he has to convert it to intermediary bytecode by using a compiler suite (such as LLVM). The next step is to identify a system event (called a "hook") to attach the program to, and then load the program into the linux kernel by using one of the available eBPF libraries. This bytecode runs in the eBPF virtual machine within the kernel. The eBPF virtual machine, like any virtual machine, is a software implementation of a computer.

4.3.2 Hooks

eBPF programs can be loaded into and removed from the kernel dynamically. Once they are attached to a hook, they'll be triggered by that event regardless of what caused that event to occur. Hooks are predefined and can include events such as network events, system calls, function entry and exit, and kernel tracepoints. If there is no predefined hook for a certain requirement, a developer can create a user or kernel probe (uprobe or kprobe). For example, a user can attach an eBPF program to the execve system call and it will be triggered whenever any process executes "execve()". It is important to mention that it doesn't matter whether that process was already running when the program was loaded. This is a huge advantage compared to upgrading the kernel and then having to reboot the machine to use its new functionality.

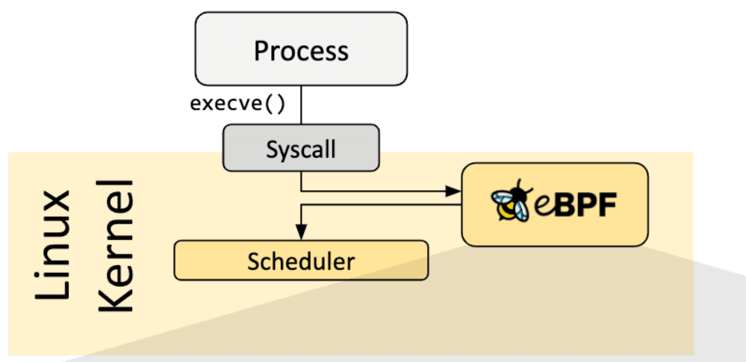

Figure 4.2: Attaching an eBPF program to a "Hook" point

4.3.3 JIT Compiler

Once loaded into the kernel, the program is automatically verified through the verification engine, and its bytecode is compiled—via a just-in-time (JIT) compiler [30]. The JIT compiler translates the generic bytecode of the program into the machine specific instruction set to optimize execution speed of the program. This makes eBPF programs run as efficiently as natively compiled kernel code or as if its code was loaded as a kernel module.

4.3.4 Verifier

An eBPF program is loaded into the kernel which means that there must be a verification process to ensure that the program is safe to be run inside the kernel [30]. Let's focus on how the verifier achieves this goal.

First of all, the verifier works on eBPF bytecode not directly on the source. That bytecode depends on the output from the compiler. Due to compiler optimization, a change in the source code might not always result in what was expected in the bytecode. As a consequence, it might not give the expected result in the verifier's verdict. For instance, the verifier rejects unreachable instructions, but the compiler might optimize them away before the verifier assesses them.

The verification step involves checking every possible execution path through the program and ensuring that every instruction is safe. The verifier, steps through the instructions in order, evaluating them rather than actually executing them. Each time the verifier comes to a branch, where a decision has to be made on whether to carry on in sequence or jump to a different instruction, the verifier pushes a copy of the current state of all the registers onto a stack and explores one of the possible paths. It continues evaluating the instructions until it reaches the return at the end of the program (or reaches the limit on the number of instructions it will process, which is currently one million instructions), at which point it pops a branch off the stack to evaluate next. If it finds an instruction that could result in an invalid operation, it fails verification.

Verifying every single possibility could get computationally expensive, so in practice there are optimizations called state pruning that avoid reevaluating paths through the program that are essentially equivalent. As it works through the program, the verifier records the state of all the registers at certain instructions within the program. If it later arrives at the same instruction with registers in a matching state, there is no need to continue to verify the rest of that path, as it's already known to be valid.

Here are the steps the verifier has to take to ensure the safety of the eBPF program :

Validating Helper Functions

A user is not allowed to call directly from eBPF programs any kernel function. Instead, eBPF provides a number of helper functions (4.5) that enable programs to access kernel information. Different helper functions are valid for different eBPF program types. For example, the helper function bpf_get_current_pid_tgid() retrieves the current userspace process ID and thread ID, but it does not make sense to call this from an XDP program that is triggered by the receipt of a packet at a network interface, because there is no userspace process involved. Also, the verifier checks that if a user is using an eBPF helper function that's licensed under GPL, his program also has a GPL-compatible license.

Checking Memory Access

The verifier performs a number of checks to ensure that eBPF programs only access memory regions that have access to.

Checking Pointers Before Dereferencing Them

One of the most common errors that cause a program to crash is to dereference a null pointer. Pointers indicate where in memory a value is being stored. Since null is not a valid memory location, the eBPF verifier requires all pointers to be checked before they are dereferenced so that this type of crash can't happen.

Accessing Context

In every eBPF program some context information is passed as an argument. Depending on the program and the attachment type it may be allowed to access only some of that context information. For example, tracepoint programs receive a pointer to some tracepoint data. The format of that data depends on the particular tracepoint. Although every tracepoint program starts with some common fields not every field is accessible from the eBPF program. Only the tracepoint-specific fields that follow can be accessed. Attempting to read or write the wrong fields leads to an invalid bpf_context access error.

Running to Completion

The verifier ensures that the eBPF program will run to completion. Otherwise, there is a risk that it might consume resources indefinitely. It does this by having a limit on the total number of instructions that it will process which is set at one million instructions at the time of this writing. That limit is hardcoded into the kernel and it's not a configurable option. If the verifier hasn't reached the end of the BPF program before it has processed this many instructions, it rejects the program.

Checking the Return Code

The return code from an eBPF program is stored in Register 0 (R0). If the program leaves R0 uninitialized, the verifier will fail.

4.3.5 Deployment of eBPF code

The eBPF code at this stage is ready to be invoked by the specified hook. Once the eBPF code is triggered, it can call special "helper" functions (4.5) that can perform a wide range of tasks, including searching and updating key-value pairs in tables, generating random numbers, redirecting network packets, and more. For security and stability reasons, these helper functions must be predefined by the kernel, but the list of helper calls available to eBPF is quite large [29]. As a result, thanks to eBPF, developers can create projects covering a wide range of use cases without having to modify kernel source code and therefore without risking the security or reliability of the kernel.

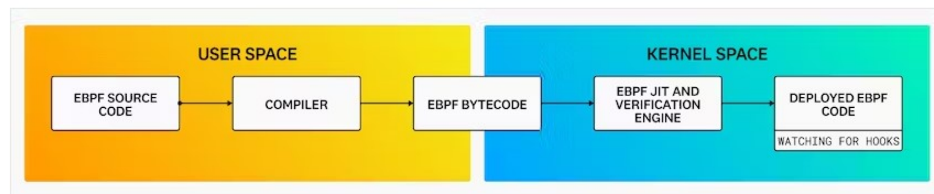

Figure 4.3: Steps to execute an eBPF program

4.4 High Performance of eBPF Programs

eBPF programs provide a very efficient way to add instrumentation to a computer system [30]. Once loaded and JIT-compiled, the program runs in high speed as native machine instructions on the CPU. Furthermore, the cost of transitioning between user and kernel space to handle each event is eliminated.

For performance tracing, another advantage of eBPF is that relevant events can be filtered within the kernel before incurring the costs of sending them to userspace. Filtering only certain network packets was, after all, the point of the original BPF implementation. Today eBPF programs can collect information about all manner of events across a system, and they can use complex and customized programmatic filters to send only the relevant subset of information to userspace.

4.5 eBPF Helpers

eBPF differs from the older cBPF in several aspects, one of them being the ability to call special functions called "helpers" from within a program [30]. These

functions are restricted to a white-list of helpers defined in the kernel.

An eBPF program cannot arbitrarily call into a kernel function. This is because eBPF programs need to maintain compatibility and avoid being bound to specific versions of the kernel. Thus, eBPF programs use helper functions to make function calls.

Helpers are APIs provided by the kernel and are used by eBPF programs to interact with the system, or with the context in which they work. For instance, they can be used to print debugging messages, to get the time since the system was booted, to interact with eBPF maps, or to manipulate network packets. Due to the fact that there are several eBPF program types that they do not run in the same context, each program type can only call a subset of those helpers. In addition, due to eBPF conventions, a helper can not have more than five arguments.

Internally, eBPF programs call directly into the compiled helper functions without requiring any foreign-function interface. As a result, calling helpers introduces no overhead, thus offering excellent performance.

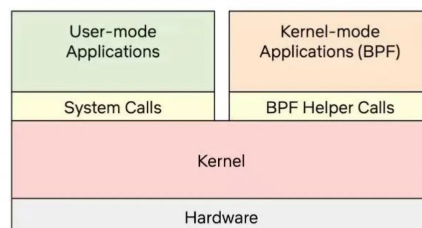

Figure 4.4: Modern Linux : a new OS model

4.6 eBPF Maps

eBPF Maps are one of the most significant features that distinguish eBPF from cBPF [30]. Maps are data structures that can be used to share information among multiple eBPF programs or to pass messages between a userspace application and eBPF code running in the kernel.

In general, all maps have the form of a key–value store. eBPF provides different types of maps such as arrays which always have a 4-byte index as the key type, hash tables that can use some arbitrary data type as the key, first-in-first-out queues, first-in-last-out stacks, least-recently-used data storage, longest-prefix matching, and Bloom filters (a probabilistic data structure designed to provide very fast results on whether an element exists).

Some map types have per-CPU variants, which means that the kernel uses a different block of memory for each CPU core's version of that map. On the other hand, for the non per-CPU variants a spin lock support was added to ensure that multiple CPU cores can access the same map while establishing concurrency.

Typical uses include the following:

- A userspace application stores information to be retrieved by an eBPF program.
- An eBPF program can use maps to store its current state so that it can be retrieved by another eBPF program (or from the same program in a future run).
- An eBPF program stores its results into a map enabling a userspace application to retrieve them for further exploitation.

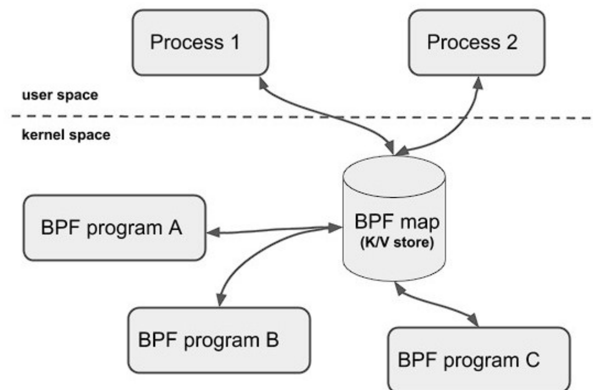

Figure 4.5: Abstract image of eBPF maps

4.7 eBPF in Page Cache

Up to this point, we've examined how eBPF works and what are some of its most common uses. However, we've yet to mention how to exploit this new technology to address our primary objective; which is how to mitigate the linux kernel overhead. In order to achieve our goal we must take a step back and focus our attention on the Page Cache.

We've mentioned in Chapter 3, that the Page Cache is a software component with a crucial role in the overall system's performance, serving as the intermediary between a storage device and a CPU's hidden cache. In addition, we've covered the Page Cache Read-Ahead mechanism for fetching pages from a storage device in order to reduce system latency. These methods adopted from the Page Cache have a general use purpose. However, better results could be achieved if users could define what pages should be fetched from a storage device in order to match the patterns of their userspace applications. For that purpose, we propose to create

an eBPF program that will allow users to determine themselves what pages of a file will be added to the Page Cache before the first Page Cache miss occurs.

Currently, eBPF for safety reasons, prohibits users from directly modifying the source code of the linux kernel or its routines. Consequently, its strict rules and policies do not permit users to make alterations on the way the Page Cache operates. In response to this limitation, we introduce a new BPF Helper under the name of "bpf_force_page2cache()" that provides the necessary means to allow user-defined pages to be added in the Page Cache.

4.7.1 bpf_force_page2cache()

Every eBPF program consists of two parts. The first part belongs to userspace and is responsible for executing the system calls that will open the eBPF application, load eBPF programs, send them to the Verifier, and finally attach them to the tracepoint handler. Users can efficiently perform these operations by leveraging eBPF libraries such as libbpf [31] that provide the necessary functions to complete these steps. Furthermore, from the userspace part of the eBPF program, users can create an eBPF map and store values in it. We propose to users to take advantage of this capability offered by eBPF and create an array-type eBPF map in which they can store the page offsets of the file that they wish to be added to the Page Cache.

The second part belongs to kernel space and is the part of the eBPF program that will interact with the linux kernel. The kernel part of the eBPF program has the capability to interact with the eBPF helper functions and as a result it is tasked with handling the "bpf_force_page2cache()" function. It is important to mention that in order to achieve maximum performance for a file operation, such as read() or mmap(), the pages of the file must be added to the Page Cache before the first Page Cache Miss occurs. Therefore, we attach our eBPF program to the kernel function that is responsible for reading pages from the Page Cache which is "filemap_get_pages" [32]. At this hook point, we can access all necessary kernel data structures needed to equip the "bpf_force_page2cache()" function so that it can add pages to the Page Cache. Additionally, since users have previously stored the page offsets in an eBPF map the "bpf_force_page2cache()" is ready to be executed.

We've created our new eBPF helper "bpf_force_page2cache()" with two input arguments. The first argument is the kernel data structure "file" [33] which is responsible for storing essential information required by the kernel for a file operation. Specifically, we exploit the "file" structure in order to define a "readahead_control" kernel data structure [34], that is responsible for issuing Read-Ahead requests for consecutive pages.

The second argument of the function is the eBPF map in which the user has stored the page offsets that will be added to the Page Cache. Because eBPF data structures require specific headers, the eBPF helper goes through the map and

stores the page offsets in an array. As illustrated in the helper's source code 4.1, the first item stored in the eBPF helper is the number of offsets the user wishes to add to the Page Cache, and the rest of the map contains the corresponding page offset values.

"bpf_force_page2cache()" is an eBPF helper and therefore it is also an integral part of the linux kernel which means that it can interact with every other kernel function. Given the complexity of the task at hand, we've introduced a new kernel function that our new eBPF helper will call, with the name "offload_pages2cache", that can actually read pages from a storage device and store them to the Page Cache. Essentially, "bpf_force_page2cache()" reads the page offsets that are stored in the eBPF map and invokes "offload_pages2cache" providing it with the user-defined offsets and the kernel structures it requires to accomplish its task. After the execution of "bpf_force_page2cache()" there are two possible outcomes. Either the pages will be successfully added to the Page Cache and the eBPF program we terminate its execution or an error will occur causing the linux kernel to take over control and execute the requested file operation. Either way our program will not cause any harm to the kernel or crash the execution of the file operation.

```
1  BPF_CALL_2(bpf_force_page2cache, struct file **, f, struct bpf_map *, map)
2  {
3    unsigned long i = 0;
4
5    WARN_ON_ONCE(!rcu_read_lock_held() && !rcu_read_lock_bh_held());
6    unsigned long *nr_pages =(unsigned long)map->ops->map_lookup_elem(map,&i);
7    int *indexes = kzalloc(*nr_pages*sizeof(int), GFP_ATOMIC);
8    if (nr_pages != NULL)
9    {
10     for(i=1; i <= *nr_pages; i++)
11     {
12       WARN_ON_ONCE(!rcu_read_lock_held() && !rcu_read_lock_bh_held());
13       unsigned long *index=(unsigned long)map->ops->map_lookup_elem(map,&i);
14       indexes[i-1] = *index;
15     }
16
17   }
18
19   struct file *filp = *f;
20   struct address_space *mapping = filp->f_mapping;
21   struct file_ra_state *ra = &filp->f_ra;
22
23   DEFINE_READAHEAD(ractl, filp, ra, mapping, 0);
24
25   offload_pages2cache(&ractl, *nr_pages, indexes);
26
27   kfree(indexes);
28   return 0;
29 }
```

Listing 4.1: source code of bpf_force_page2cache() eBPF helper

4.7.2 offload_pages2cache()

"offload_pages2cache()" is the kernel function we've developed, that is responsible for reading pages into the Page Cache. The function, starts with validating the input pointers, to ensure they are not NULL. Its main objective is to loop over the user-specified pages and identify consecutive sequences in order to initiate Read-Ahead requests for each sequence. The function must also handle gaps in the sequence or cases in which a page extends beyond the end of the file. Notably, the actual Read-Ahead process is executed by invoking the "page_cache_ra_unbounded" [35] kernel function with the calculated number of pages to read.

```
void offload_pages2cache(struct readahead_control *ractl, unsigned long
    nr_to_read, int *indexes) {
  if(indexes == NULL) {
    printk(KERN_DEBUG "offload_pages2cache indexes == NULL return...");
    return ;
  }
  if(ractl == NULL) {
    printk(KERN_DEBUG "offload_pages2cache ractl == NULL return...");
    return ;
  }
  struct inode *inode = ractl->mapping->host;
  unsigned long index, prev_index, i, j, seq_pages_to_read;
  loff_t isize = i_size_read(inode);
  pgoff_t end_index;  /* The last page we want to read */
  if (isize == 0)
    return;
  end_index = (isize - 1) >> PAGE_SHIFT;
  for(i=0; i<nr_to_read; i++)
  {
    seq_pages_to_read = 0;
    index = indexes[i];
    ractl->_index = index;
    if (index > end_index)
      return ;
    seq_pages_to_read += 1;
    for(j = i + 1; j < nr_to_read; j++)
    {
      prev_index = index;
      index = indexes[j];
      if (index > end_index)
        return page_cache_ra_unbounded(ractl, seq_pages_to_read, 0);
      if (index != prev_index + 1)
        break;
      seq_pages_to_read += 1;
      // Don't read past the page containing the last byte of the file
      if (seq_pages_to_read > end_index - ractl->_index)
      {
        seq_pages_to_read = end_index - ractl->_index + 1;
        return page_cache_ra_unbounded(ractl, seq_pages_to_read, 0);
      }
    }
    i = i + seq_pages_to_read - 1;
    page_cache_ra_unbounded(ractl, seq_pages_to_read, 0);
  }
}
```

Listing 4.2: source code of offload_pages2cache() kernel function

4.7.3 bpf_get_filename()

The linux kernel is a multitask O.S. and therefore there could be a number of file operations being executed at the same time. However, a user is interested in the file operation that is referring to a specific file and as a result he must be able to specify that in the eBPF program. For that purpose, we've developed a second eBPF helper, with the name "bpf_get_filename()", that is responsible with handling this task. "bpf_get_filename()" has three input arguments :

1. the filename the user is interested in.
2. the size of the filename (for safety reasons).
3. The kernel data structure from which the eBPF helper can retrieve the filename that corresponds to the current process (and which the helper will compare with the filename that the user specified).

"bpf_get_filename()" has a return value of either 0 or 1. If the filename of the current process is the same with the filename that the user specified then the eBPF helper returns 1 otherwise it returns 0.

"bpf_get_filename()" is an eBPF helper and therefore it is executed in the kernel side of the eBPF program. In addition, from the hook that the eBPF program is attached to ("filemap_get_pages") we can retrieve the necessary kernel data structure ("file") the helper requires for its execution. As a result, the user is only obliged to specify the filename that is interested in.

```
1  BPF_CALL_3(bpf_get_filename, char *, filename, u32, size, struct file **, f)
2  {
3    if (unlikely(!f))
4      goto err_clear;
5
6    struct file *filp = *f;
7    struct path *f_path = &filp->f_path;
8    struct dentry *dentry = f_path->dentry;
9    const struct qstr *dname = &dentry->d_name;
10
11   int ret = strncmp(filename, dname->name, size);
12   if(ret != 0)
13     return 0;
14   return 1;
15 err_clear:
16   return 0;
17 }
```

Listing 4.3: source code of bpf_get_filename() eBPF helper

Chapter 5

Experimental evaluation

5.1 Simulation Tool : fio - Flexible I/O tester

We employ the FIO simulation tool [36] to conduct our experiments and measurements. FIO, which stands for Flexible I/O, is a third-party tool designed to simulate specific I/O workloads. It allows us to quickly define and execute workloads, providing detailed metrics for each run. This capability enables us to compare results across various conditions, including different hardware and firmware configurations, using workloads that closely resemble production scenarios.

The workload gives the capability to test various I/O scenarios that represent real-life usage patterns on computer systems. For example, it enables testing sequential read and write operations as well as random read and write operations. It also enables the ability to test single-threaded vs. multi-threaded I/O operations as well as the ability to control I/O queue depths and whether hardware caches should be used.

In the following sections we present results of :

- fio tests over a file that is currently located only on a storage device and not on the Page Cache.
- two types of file operations :
  - read()
  - mmap()
- A single thread is executing the file operations.
- two types of access patterns :
  - sequential access patterns.
  - random access patterns.

5.2 Page Cache

Up to this point we've examined the Page Cache and its mechanisms. We've stated before that the Page Cache Read-Ahead mechanism is very efficient when it comes to sequential access patterns and behaves poorly when it comes to random access patterns. In this section, we will dive into the Page Cache efficiency and present our results. In all our experiments we will focus our attention on the file operations latency.

5.2.1 Latency

Latency is the flip side of the same performance coin. Where throughput refers to how many bytes of data per second you can move on or off the disk, latency—most commonly measured in milliseconds—refers to the amount of time it takes to read or write a single block. In fact, storage bottlenecks are actually latency issues that affect throughput, not the other way around.

We can understand how latency affects throughput from a simple experiment. If we have a reasonably fast disk with a maximum throughput of 180MB/sec and a total access latency of 16ms, and we present it with a maximally fragmented workload—meaning that no two blocks have been written/are being written in sequential order—we can do a little math to come up with that throughput. Assuming 4KB physical blocks on disk, 4KB per seek divided by 0.016 seconds per seek = only 250KB/sec, a disappointing performance.

5.2.2 read()

This section explores the impact of a sequential and a random access pattern on the latency of the read() file operation. The effectiveness of a sequential read() comes from the fact that the process is accessing file pages consecutively and thus leading the linux kernel to consistently hit the PG_Readahead marker and prefetch file pages before they are actually requested. Consequently, we expect that only one Page Cache miss will occur and therefore the performance for a sequential read() will depend on the disks performance and the variation of the Read-Ahead window size. Conversely, in random read() operations, the kernel cannot predict the pages the process will request and thus can not correctly mark pages with the PG_Readahead flag, making it challenging to predict the effectiveness of the Read-Ahead mechanism.

The following figure 5.1 illustrates the results for both sequential and random read() operations, highlighting the expected outcome of only one Page Cache miss for sequential reads and varying numbers of Page Cache misses for random reads. It is important to mention that there is a significant difference in the number of Page Cache misses between the two access patterns, particularly evident in the results for a 2GB file, where a substantial contrast is exhibited.

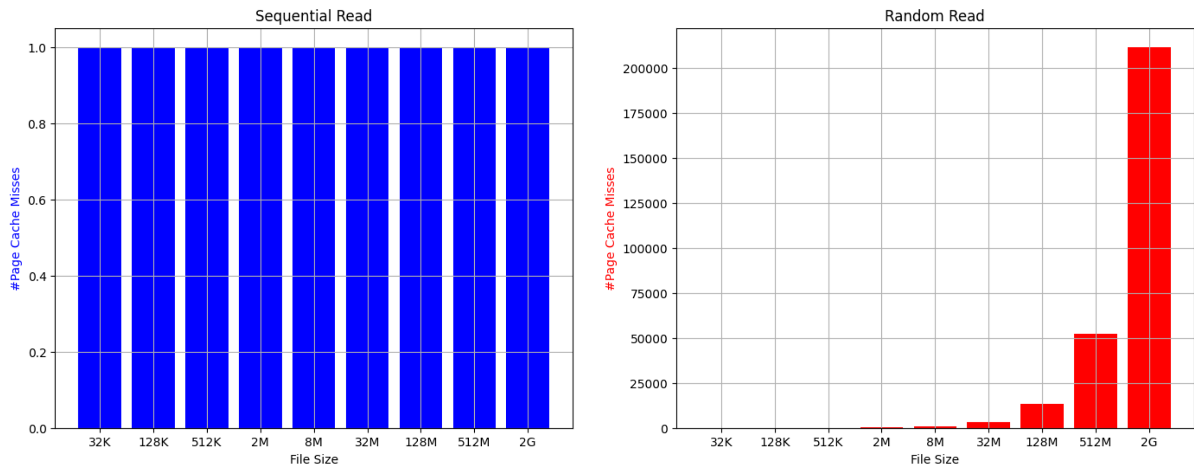

Figure 5.1: Number of Page Cache Misses for read() operations with Sequential (Left) and Random (Right) access patterns.

In Chapter 3, we've examined the complexity of retrieving data stored in a disk. Consequently, an elevated number of Page Cache misses, which results in more requests to a storage device, leads to an increase in the latency of file operations. In the following Figure 5.2, we showcase the completion time of each file operation that corresponds to what we've presented in Figure 5.1.

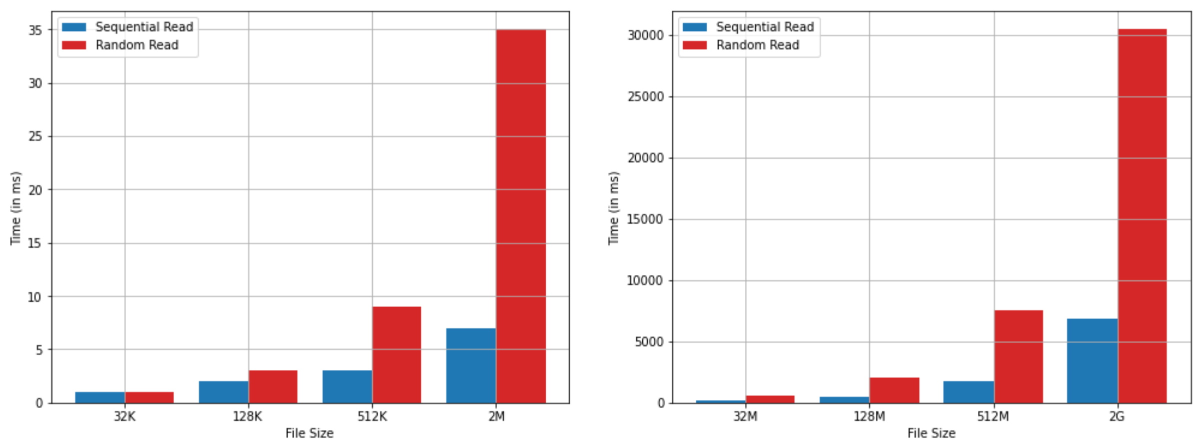

Figure 5.2: Latency for Sequential and Random read() file operations. It's important to notice the significant scaling difference between the y-axis of the left and right figure.

The above results illustrate what we've mentioned so far providing conclusive data that showcase the significant difference in the performance of the Page Cache when dealing with processes with more complex access patterns. In almost every

examined case, the latency for random access pattern is two or three times more than that of the sequential access pattern.

5.2.3 mmap()

Everything we've showcased so far for the read() file operation also stands for the mmap() file operation. In fact, as we can observe from the figure 5.3 the performance gap between sequential and random access patterns for the mmap() file operation is much more substantial than it was for the read() file operation.

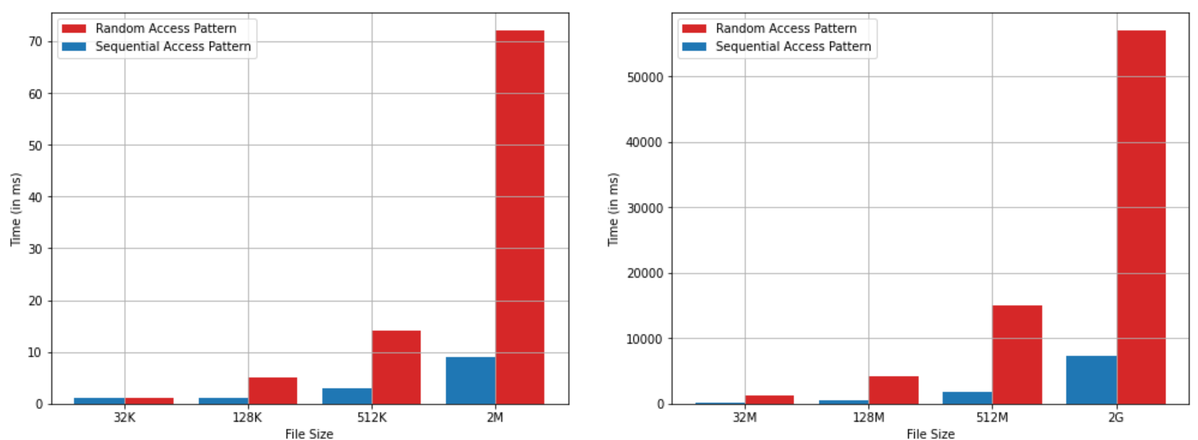

Figure 5.3: Latency of mmap() file operations with Sequential and Random Access Pattern. It's important to notice the significant scaling difference between the y-axis of the left and right figure.

5.3 Impact of "bpf_force_page2cache()" in Page Cache

In the previous cases, the user is aware that the process will access every page of the file. Therefore, it is time to deploy "bpf_force_page2cache()" to add pages to the Page Cache. The following figures present the influence of the new eBPF helper to read() and mmap() file operations.

5.3.1 read()

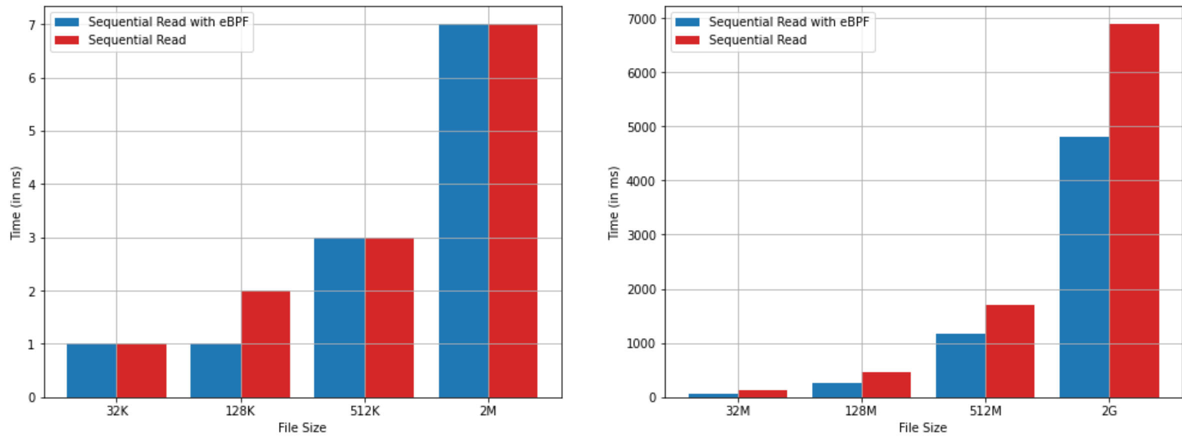

Figure 5.4: Latency of Sequential read() with and without eBPF. It's important to notice the significant scaling difference between the y-axis of the left and right figure.

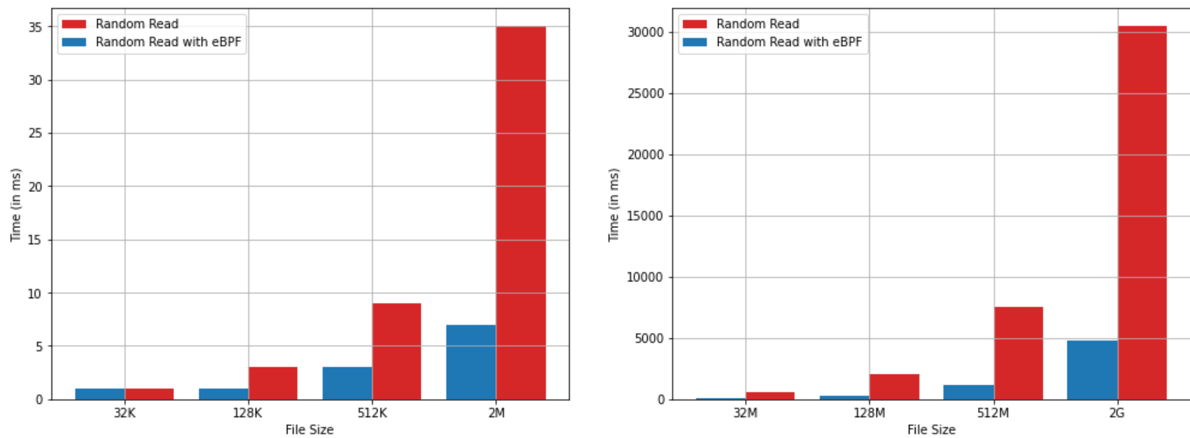

Figure 5.5: Latency of Random read() with and without eBPF. It's important to notice the significant scaling difference between the y-axis of the left and right figure.

5.3.2 mmap()

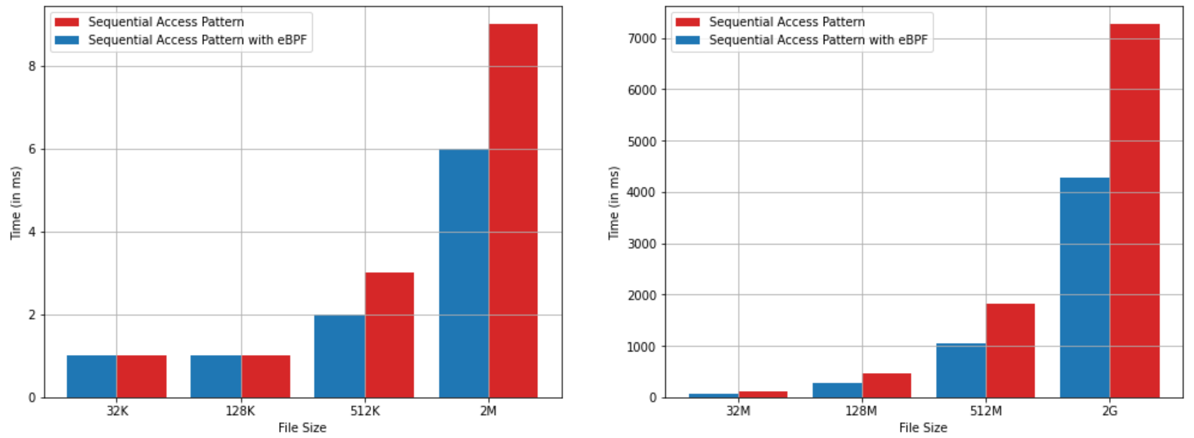

Figure 5.6: Latency of mmap() for Sequential Access Pattern with and without eBPF. It's important to notice the significant scaling difference between the y-axis of the left and right figure.

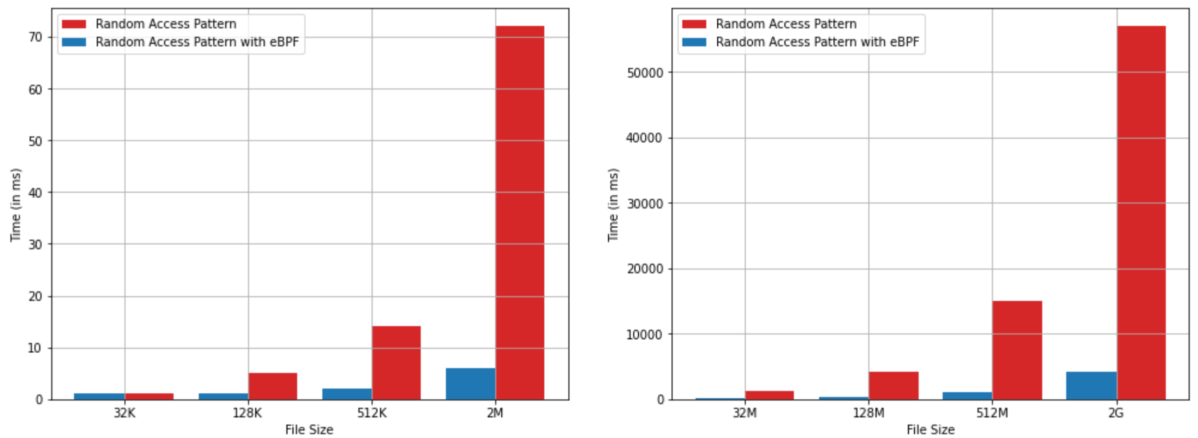

Figure 5.7: Latency of mmap() for Random Access Pattern with and without eBPF. It's important to notice the significant scaling difference between the y-axis of the left and right figure.

5.3.3 Results

Sequential Access Pattern

In the case of sequential access patterns, the influence of the eBPF helper on the performance of file operations depends on the file size. For smaller file sizes, the eBPF helper exhibits a natural impact, neither deteriorating nor significantly enhancing the file operations performance. However, as the file sizes increase, the impact of the eBPF helper becomes noticeable, indicating its utility in larger files.

Random Access Pattern

In the case of random access patterns, the impact of the eBPF helper on file operations latency is immediately evident. The results clearly prove that the utilization of the eBPF helper has reduced the latency for both file operations, read() and mmap(), to levels comparable to the corresponding latency of the sequential access patterns. Consequently, in this case, we have achieved our goal of minimizing the performance gap between the two types of access patterns.

5.4 Use Case : Firecracker

Firecracker [38] is a virtual machine monitor (VMM) that uses the linux kernel-based Virtual Machine (KVM) to create and manage microVMs. Firecracker was primarily developed as a specialized VMM for serverless workloads, but it is generally useful for containers, functions and other compute workloads. It has a minimalist design that intentionally excludes unnecessary devices and guest functionality, aiming to reduce the memory footprint of each microVM and leading to reduced startup times and increased hardware utilization. With firecracker, cloud providers can pack thousands of microVMs onto the same machine. This means that every serverless function, container, or container group can be encapsulated with a virtual machine barrier, enabling workloads from different customers to run on the same machine, without any tradeoffs to security or efficiency.

5.4.1 Serverless computing

Serverless computing [39] has emerged as the fastest growing cloud service and deployment model of the past few years. In serverless, services are decomposed into collections of independent stateless functions that are invoked by events specified by the developer. These functions are very popular due to their reduced cost of operations, improved utilization of hardware, and faster scaling than traditional deployment methods. Consequently, at any given time there could be a number of active functions concurrently running that could range from zero to thousands. Moreover, the serverless model provides a pay-as-you-go billing allowing customers to be charged only for the time spent executing their requests.

5.4.2 Cold Start Problem

The economics and scale of serverless applications demand that workloads from thousands of independent function instances run on the same hardware with minimal overhead, while preserving strong security and performance isolation. This high degree of co-location has proven to be possible from a recent study of Azure Functions [40] in production. The study shows that serverless functions are short-running and invoked infrequently. Specifically, it shows that half of the functions complete within 1 second while >90% of functions have runtime below 10 seconds. Another finding is that functions tend to have small memory footprints: >90% of functions allocate less than 300MB of virtual memory. Lastly, 90% of functions are invoked less frequently than once per minute, albeit >96% functions are invoked at least once per week.

We are interested in the Azure study because the process inside the firecracker's microVM, which receives the function invocation in the form of an RPC, takes up to several seconds to bootstrap before it is able to invoke the user provided function, which also may have its own initialization phase [39]. Considering the short execution time of serverless functions that the Azure study revealed, the period it takes to initialize them, commonly referred to as "cold start", constitutes a significant portion of the total execution time and introduces a notably expensive latency. In addition, customers are not billed for the time a function boots and have a strong incentive to minimize cold starts due to their impact on latency.

In order to avoid this problem, both cloud vendors and their customers prefer to keep function instances memory resident, commonly referred to as keeping them "warm". However, cloud vendors can not predict whether a customer will execute the same function again and therefore keeping idle function instances alive could turn out to be a waste of precious main memory. As a result, given that serverless providers deploy thousands of functions on a single server, the memory footprint of keeping all instances warm can extend into hundreds of gigabytes. Another thing to keep in mind, is that a servers main memory accounts for a large portion of its typical capital cost and thus serverless providers can't afford to waste it.

To avoid unnecessary memory usage, most serverless providers tend to limit the lifetime of function instances to 8-20 minutes after the last invocation, due to the sporadic nature of invocations. In other words, providers prefer to remove instances after a period of inactivity and start new ones on demand. In the last few years, high cold-start latencies have become one of the central problems in serverless computing and one of the key metrics for evaluating serverless providers.

5.4.3 Snapshotting

Snapshotting is an innovative technique proposed by researchers to help reduce cold starts while avoiding the need to keep thousands of functions warm [39]. This method aims to quickly restore a virtual machine (VM) to its warm state by capturing it to a snapshot. Specifically, a snapshot captures the current state of a VM, including the state of the virtual machine monitor (VMM) and the guest-physical memory contents, and store it as files on disk. With snapshotting, we can capture the state of a function instance that has been fully booted and is ready to receive and execute a function invocation and therefore we no longer need to keep it alive. Upon the next invocation of the function, a new instance can be quickly created from the corresponding snapshot. After the loading process is complete, this instance is ready to process the incoming request, effectively eliminating the high cold start latency.

Firecracker has introduced their own open-source snapshotting mechanism [41] that follows the same design principles as Catalyzer [39]. Similarly to Catalyzer, loading a Firecracker VM from a snapshot is done in two phases. First, the hypervisor process loads the state of the VMM and the emulated devices (that we further refer to as loading VMM for brevity) and then maps a plain guest-physical memory for lazy paging.

Snapshotting Latency issue

In order to avoid loading the whole guest memory when starting guest VMs, snapshot methods apply lazy loading of guest memory [42]. With Lazy loading memory pages are loaded from disk on-demand when accessed by the guest. However this proves to have a negative impact on the snapshotting technique because the page access pattern tends to be closer to random than sequential and exhibit low spacial locality. Therefore, this leads to many major page faults and scattered disk reads that add significant overhead and slow down function execution. As a result, although snapshot and restore improve performance, the cold start problem is still not entirely solved. This is because in order to restore the VM state and initialize the guest memory, the guest needs to access at least a few thousands of memory pages with lazy loading.

5.4.4 Deployment of eBPF

In order to enhance the performance of the snapshotting technique we can leverage our new eBPF helper. To achieve our goal, in the first invocation of restoring a snapshot we will record the pages accessed and store them into a file. Next, we will assume that memory accesses are stable across function invocations and use the stored information to prefetch a compact representation of the working set of previous invocations when handling new ones. Finally, we will leverage "bpf_force_page2cache()" to bring the user-defined pages into the Page Cache through a single read(), thus avoiding the costs associated with scattered page reads and reduce the techniques latency.

5.4.5 Results

In the following figure 5.8 we present the impact of "bpf_force_page2cache()" to firecracker's snapshotting. The following results occur from loading a snapshot, using it to resume a firecracker's microVM and executing a simple function.

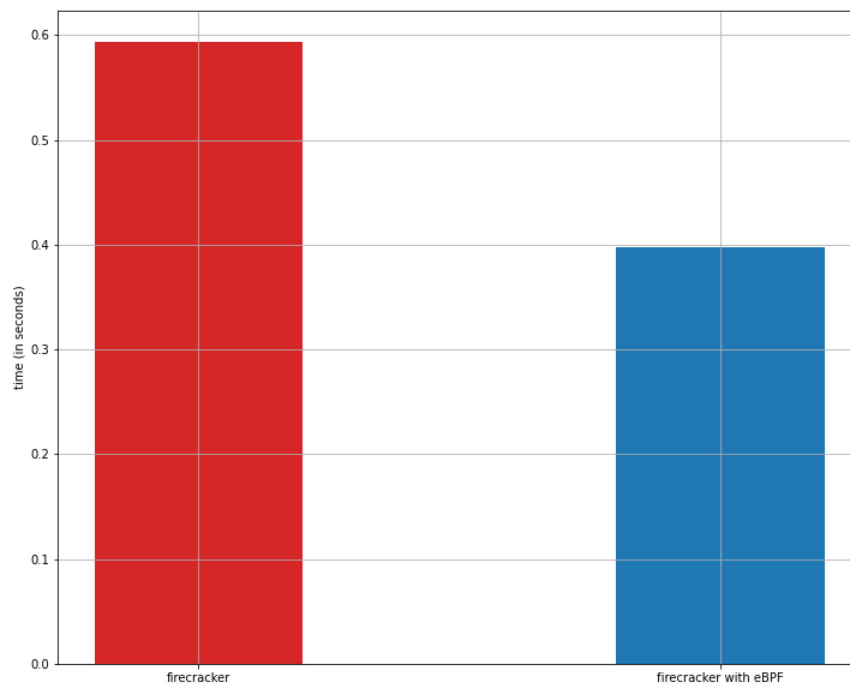

Figure 5.8: Results of eBPFs impact on firecracker's snapshotting method

On the left bar, we present the operations latency for firecracker's snapshotting method and on the right bar the corresponding latency for the same operation with the influence of our new eBPF helper.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this Diploma thesis, we've conducted a comprehensive examination of the software layers and components of the linux kernel that are associated with the read file operation. Furthermore, we've delved into the Page Cache and its key roll on the file operation's latency. Specifically, through a thorough analysis of the Page Cache, we've demonstrated that :

- Processes with sequential access patterns have an ideal performance. This outcome come from the fact that the Page Cache Read-Ahead mechanism manages to pre-fetch pages before they are actually requested. Thus, it eliminates the time it would normally take for the process to request a set of pages every time a Page Cache miss occurred.
- Processes with complex access patterns have a significant higher latency compared to those with sequential access. The main reason for this consequence is the inadequacy of the Page Cache Read-Ahead algorithm to adapt to access patterns that are more complex and seem to be random. As a result, a sufficient number of Page Cache misses occurs, that introduce additional latency to the completion of file operations.

Motivated by the unsustainable performance of the Page Cache for processes with complex access patterns, we've shifted our attention to a new powerful technology of the linux kernel named eBPF to find a solution. We've examined how the evolution of eBPF has led it to be a tool used to safely and efficiently extend the capabilities of the kernel without requiring to change kernel source code or load kernel modules. We've leveraged eBPF to create a new helper that will allow user to define the pages that will be added to the Page Cache in order to match the access patterns of their userspace applications.

More information about the development of the "bpf_force_page2cache()" eBPF helper can be found on this Github repository : [43].

From our tests and results we've conducted the following conclusions :

- In order to develop a new eBPF helper we've modified the linux kernel. However, we've successfully completed our goal without making any major changes to the linux kernel but with only a few extra lines of code. In addition, we've kept the full support of the linux kernel when executing our new helper to add pages to the Page Cache.

- Page Cache misses lead to increased latency of file operations due to the large number of read system calls that must be handled from the kernel. To address the disparity between processes with a high number of Page Cache misses (random access patterns) and those with a small number (sequential access patterns), we utilize our new eBPF helper to efficiently fetch the pages that a userspace application will request in a single read batch. The figures presented in Chapter 5 illustrate how this approach effectively narrows the performance gap between them.

- We can take advantage of our new eBPF helper for file operations such as read() and mmap(). This capability can be utilized in applications that perform these types of operations. Specifically, we've demonstrated the utility of our eBPF helper in enhancing the snapshotting technique of firecracker, which executes the mmap() file operation to load and restore a snapshot file.

6.2 Future Work

Based on the results and conclusions we've presented, we believe that the new eBPF helper "bpf_force_page2cache()" is capable of offering significant performance improvement under certain conditions and is worth studying and evolving. Particularly, as a continuation of the present work, we would find it interesting and challenging to add these utilities:

- Currently, the eBPF helper performs synchronous read. That means that it is designed to read a batch of pages from the storage device and fetch them to the Page Cache in order to complete its operation. However, its performance could be significantly improved by implementing a hybrid approach. Specifically, it would be advantageous for the eBPF helper to read a small set of pages (e.g., four pages) synchronously, while fetching the remaining pages asynchronously in batches. This would minimize the process waiting time, as it would only need to wait for a short period until the first batch of pages arrives. By the time the process finishes reading these pages, the next batch of pages would already be available. In essence, it would be beneficial to implement a Read-Ahead mechanism for the eBPF helper.

- The next step involves incorporating multiple kernel threads into the procedure. Given that the user has predefined the pages to be added to the Page Cache and, as mentioned earlier, a hybrid approach with synchronous and asynchronous page fetching has been established, we can divide the procedure into distinct parts and assign a dedicated thread to handle each part. Specifically, the initial thread would address the synchronous segment, while concurrently, the remaining threads —whether they're one, two, or more— would asynchronously handle the retrieval of a number of additional pages each (i.e. eight), continuing this process until all specified pages have been successfully fetched into the Page Cache. This threaded implementation aims to enhance parallelism and overall efficiency in the execution of the eBPF helper operation.

Bibliography

[1] INTEL. Breakthrough performance for demanding storage workloads.
https://ark.intel.com/content/www/us/en/ark/

[2] Polling and Interrupts.
http://www.linux-tutorial.info/?page_id=418

[3] Fast Scatter-Gather I/O.
https://www.gnu.org/software/libc/manual/html_node/Scatter_002dGather.html

[4] KIM, H.-J., LEE, Y.-S., AND KIM, J.-S. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '16) (Denver, CO, USA, 2016).

[5] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting linux storage stack for µs latency and high throughput. In 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), pages 113–128. USENIX Association, July 2021.

[6] YANG, Z., HARRIS, J. R., WALKER, B., VERKAMP, D., LIU, C., CHANG, C., CAO, G., STERN, J., VERMA, V., AND PAUL, L. E. SPDK: A development kit to build high performance storage applications. In IEEE International Conference on Cloud Computing Technology and Science (CloudCom '17) (Hong Kong, 2017), pp. 154–161.

[7] Wikipedia : Locality of reference.
https://en.wikipedia.org/wiki/Locality_of_reference

[8] Wikipedia : Cache replacement policies.
https://en.wikipedia.org/wiki/Cache_replacement_policies

[9] Viacheslav Fedorov, Jinchun Kim, Mian Qin, Paul V Gratz, and AL Narasimha Reddy. Speculative Paging for Future NVM Storage. In Proceedings of the International Symposium on Memory Systems (MEMSYS), 2017.

[10] Wikipedia : PageRank.
https://en.wikipedia.org/wiki/PageRank

[11] Linux. fadvise(1) — Linux manual page.
https://man7.org/linux/man-pages/man1/fadvise.1.html

[12] Linux. madvise(2) — Linux manual page.
https://man7.org/linux/man-pages/man2/madvise.2.html

[13] Stephen Macke - GitHub. smacke/jaydio: A Java Library to Perform Direct I/O in Linux, Bypassing File Page Cache.
https://github.com/smacke/jaydio

[14] EighteenZi - GitHub. Direct-IO.md.
https://github.com/EighteenZi/rocksdb_wiki/blob/master/Direct-IO.md

[15] Wikipedia : Berkeley Packet Filter.
https://en.wikipedia.org/wiki/Berkeley_Packet_Filter

[16] LWN : A thorough introduction to eBPF.
https://lwn.net/Articles/740157/

[17] XRP: In-Kernel Storage Functions with eBPF.

[18] Wkipedia : CPU cache.
https://en.wikipedia.org/wiki/CPU_cache#

[19] Wikipedia : System call.
https://en.wikipedia.org/wiki/System_call

[20] intro(2) — Linux manual page.
https://man7.org/linux/man-pages/man2/intro.2.html

[21] Understanding the Linux Kernel, 3rd Edition, By Daniel P. Bovet, Marco Cesati, November 2005

[22] Halo Linux Services : Page Cache Readahead.
https://www.halolinux.us/kernel-architecture/page-cache-readahead.html

[23] Wikipedia : eBPF.
https://en.wikipedia.org/wiki/EBPF

[24] eBPF Explained: Use Cases, Concepts, and Architecture.
https://www.tigera.io/learn/guides/ebpf/

[25] IBM : What is eBPF? .
https://www.ibm.com/topics/ebpf

[26] Steven McCanne, Van Jacobson. The BSD Packet Filter : A New Architecture for User-level Packet Capture, 1992.

[27] Alexei Starovoitov's, BPF – in-kernel virtual machine, 2015.
https://netdevconf.info//0.1/docs/starovoitov-bpf_netdev01_2015feb13.pdf

[28] Kernel Documentation : Kernel Probes (Kprobes).
https://www.kernel.org/doc/html/latest/trace/kprobes.html

[29] Linux. bpf-helpers(7) — Linux manual page.
https://man7.org/linux/man-pages/man7/bpf-helpers.7.html

[30] Liz Rice, Learning eBPF : Programming the Linux Kernel for Enhanced Observability, Networking, and Security, March 2023.

[31] Github : libbpf.
https://github.com/libbpf/libbpf

[32] linux kernel v5.15.19 : function filemap_get_pages.
https://elixir.bootlin.com/linux/v5.15.19/source/mm/filemap.c#L2524

[33] linux kernel v5.15.19 : struct file.
https://elixir.bootlin.com/linux/v5.15.19/source/include/linux/fs.h#L965

[34] linux kernel v5.15.19 : struct readahead_control.
https://elixir.bootlin.com/linux/v5.15.19/source/include/linux/pagemap.h#L820

[35] linux kernel v5.15.19 : page_cache_ra_unbounded kernel function.
https://elixir.bootlin.com/linux/v5.15.19/source/mm/readahead.c#L174

[36] fio : Flexible I/O tester.
https://fio.readthedocs.io/en/latest/fio_doc.html#overview-and-history

[37] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20).
https://www.usenix.org/system/files/nsdi20-paper-agache.pdf

[38] Firecracker.
https://firecracker-microvm.github.io/

[39] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, Boris Grot. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. ASPLOS '21, April 19–23, 2021, Virtual, USA.
https://marioskogias.github.io/docs/reap.pdf

[40] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In Proceedings of the 2020 USENIX Annual Technical Conference (ATC). 205-218.

[41] Github : firecracker snapshotting.
https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md/

[42] Lixiang Ao, George Porter, Geoffrey M. Voelker. FaaSnap: FaaS Made Fast Using Snapshot-based VMs. EuroSys '22, April 5–8, 2022, RENNES, France.

[43] Github Repository.
https://github.com/el18053/Diploma