XRP: In kernel Storage Functions with eBPF

Yi He
Technical University of Munich

Hanwen Liu
Technical University of Munich

Wonbang Seo
Technical University of Munich

Yiwen Liu
Technical University of Munich

Abstract

With recent improvements of the NVMe devices, the software storage stack in the Linux kernel has become a major bottleneck of disk I/O. XRP is a new approach to bypass it to achieve higher I/O performance using eBPF. In this paper, we focus on analyzing the existing XRP project and improving its performance as well as functionality.

Original Paper Summary and Evaluation

1 Introduction

With the rise of the high performance memory devices, overheads in linux kernel storage stack get more significant degradations in latency and IOPS. To illustrate this, an experiement was conducted to calculate Kernel's latency overhead with 512B random reads in Intel Optane SSD P5800X. The kernel side overhead accounts for 48.6 percent of the latecy [19]. The result shows that kernel stack overhead is the major bottleneck that should be dealt with.

1.1 Kernel bypass

There has been a lot of proposals to reduce the kernelside overheads. One approach is to bypass the kernel. A Kernel bypass would theoretically reduce the overheads by the half. However, the kernel bypass brings about low granurarity, ineffient I/O polling, and the CPU contention. And the problems led an unexpected degradation in latencies and throughput.

1.2 eBPF

eBPF (extended Berkeley Packet Filter) [1] is an approach to tackle the problems that arise in kernel bypass. eBPF is an interface with which users can offload simple functions to any kernel layers.

eBPF benefits legacy reads by cutting down overheads traversing the majority of kernel stack. It prevents a string of auxiliary I/O requests from traversing the stack by resubmitting such a request at the layer where a BPF fucntion hooked. For example, intermediate pointer lookups can be executed by a BPF fucntion in the course of B-tree search. Zhong et al. compared the performance improvement from a resubmission hook and the result showed the resubmission process yields 1.8-2.5 times throughput increase after reaching CPU-saturation. As with legacy reads, *io_uring* [3] also can benefit from eBPF. Each I/O submitted with *io_uring* do not need to passes through all the layers with the help of eBPF.

However, the resubmissions with in the NVMe drivers lack metadata, the high level of context. As the drivers do not have any access to it, a BPF function cannot hop to the next offset. In addition, concurrency cannot be guaranteed as well as NVMe layer does not access to both page cache and the information on a data structure. The challenges make the implementation difficult. That said, the observations show that the files of many engines remain relatively stable; some data structures are immutable, others are mutable but less frequent. [19]

1.3 eBPF Design Principles

The observations led the following design principles:

- **One file at a time** restrict a chain of resubmissions on a single file, which minimizes the metadata
- **Stable data structures** target the DBs that stay immutable for a long period of time
- **User-managed caches** manage blocks in user-managed caches
- **Slow path fallback** in case of a traversal fails, the application must retry or fall back to dispatching the I/O requests using user space system calls

2 XRP Design

2.1 Solution Main Ideas

To bypass most layers of the kernel and not raise the current problems, they propose the eXpress Resubmission Path(XRP). The key insights are the following two parts:

- **Use eBPF functions to offload functions in disk**

  Their solution is not directly bypassing the kernel layer, otherwise they indirectly cleverly exploit the properties of eBPF [1]. Their scheme is not simply to unuse the full kernel stack to reduce the time, they will traverse the full stack in the very first and very end steps. [19] In the intermediate steps, the BPF function can help us execute the simple functions like parsing the B-tree. So XRP can reduce the useless intermediate data and a series of the syscall from user space.

- **Resubmmission Logic**

  Unlike the common usages of BPF like packet filtering and tracing usages, they implement the BPF function in the storage devices. As we have said in the Chapter 1, the low-level NVMe driver lacks the content that higher levels provide. [19] To enable we can benefit from traversing within the storage device, only with BPF cannot fetch the file information of the next node. Although we can get the corresponding logic address offset stored in the BPF function, we need an actual physical address in the process. They use metadata digest to achieve this insight. [19] We will introduce this in 2.3.3. When we resubmit the next node, we could use this to translate the corresponding logic address into the pysical addrss of the next node.

2.2 System Design

The system includes three main blocks within the NVMe driver, namely BPF hook, metadata digest, and resubmission block. We can see this in Figure 1. The BPF hook can trigger our offload functions. The XRP maintains two queues, one is to append the completion task(CQ), and the other is to append the submission queue(SQ). [19] When we call XRP, the storage device will change into an interrupt handler state. Then we will execute the offload function. Here we should note that the user functions must be checked firstly by the Linux BPF verifier. When we want to fetch the next node, the XRP invokes the metadata digest to translate the logical address into the physical address of the next submission. According to this, we can resubmit the next NVMe command.

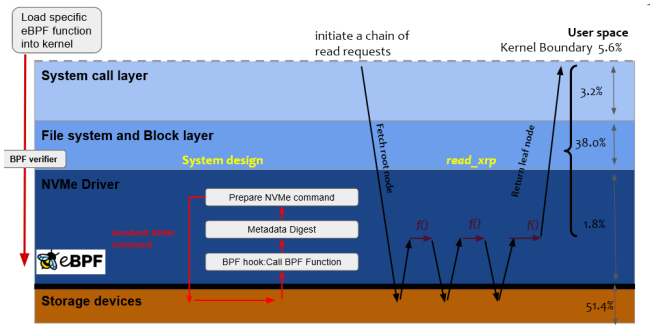

Figure 1: **System design**: The main part of XRP and an example of *read_xrp*.

2.3 Design Details

2.3.1 BPF Hook

They define a struct type *bpf_xrp* to match the offload functions. Whenever we use XRP *read_xrp* system call, we need to specify this signature so that the corresponding functions can be called by the BPF hook. This signature is shown in the listing 2.3.1. The first four fields are checked and modified by the BPF caller when we execute the resubmission logic.

`data` buffers the data read from the disk, like the content information to be parsed. `done` indicates whether we will continue the resubmission process. `next_addr` and `size` stores the next logical addressed for the next resubmission. Compared to the `data`, `scratch` is to store the process intermediate data like key values. Every time BPF function compares the intermediate key with stored information in `data`. When we find our target leaf node, it will return the key-value pair to the user space. [19]

```
1 struct bpf_xrp{
2     char *data;
3     int done;
4     uint64_t next_addr[16];
5     uint64_t size[16];
6     char *scratch
7 }
```

2.3.2 BPF Verifier

The main function of the BPF verifier is to inspect the BPF hook to ensure memory safety. The BPF verifier only accepts a scalar or a pointer value stored in each register. `Scalar` represents a value that cannot be dereferenced. And various `pointers` are defined by the verifier to check the out-of-bound access. For example, `PRO_TO_MEM` checks the referring memory region size. For XRP, the verifier will check the `data` and `scratch` fields, the first one is `PRO_TO_MEM` and the second one is `Scalar`. Additionally, each BPF function also defines a callback function *is_valid_acess()* to perform additional checks. [19]

2.3.3 The Metadata Digest

As we have said, a very significant point of XRP is resubmission logic. In the normal method, the file system will be responsible for the next read physical block physical address translation. But if we want to ensure the main process is on disk, we need to translate the logical address given by the `next_addr` field in `bpf_xrp`.

The authors design the metadata digest, which is a thin interface between the file system and the interrupt handler. The file system shares its logical-to-physical-block mapping with the interrupt handler, so when we execute the resubmission, even if the process is in the NVMe driver, we still can fetch the necessary information that we need for the next operation. [19]

One insufficiency is the metadata digest is limited by the specific file system. For each different file system, we need to design different logic to achieve this function. For example, the authors achieve the metadata digest in the ext4 file system, and the inode information is stored in a cached version of the extent status tree.

The metadata digest block contains two main functions. *update_mapping* is to keep the metadata up-to-date. The other one, *lookup_mapping*, receives the inode address and returns the corresponding physical offset and length. [19]

And the authors consider several special mechanisms at the time of implementation. First, the BPF function cannot resubmit the next logic address outside of the open file so that the query range of the lookup function is limited. Second, the XRP will perform the lookup function twice each time to avoid conflict conditions. Such as when the *update_mapping* is executed, if we read the same content meanwhile, the XRP will abort the operation. In the worse case, if the XRP detects an invalid logical address, the operation will fall back to the normal system to attempt the request again. This is what we called "slow path fallback" in the 1.3 part. [19]

2.3.4 Resubmitting NVMe Requests

Until this block, we have received the physical address of the next fetch node. So we have fulfilled the prerequisites of NVMe request. Updating the physical sector and block addresses of the existing NVMe request struct: `struct nvme_command`, according to the metadata digest. By this way, we achieve the next NVMe command and go back to the original BPF hook to call functions and start a new cycle.

2.4 Design Limitations

2.4.1 Synchronization Limitations

Due to BPF program can only obtain one lock at a time, and the programm must release the lock before returning. And the user space must through syscall to access the lock-protected structure. So we cannot offload complex functions that require several synchronization. [19]

2.4.2 Scheduler Limitations

The author observes that the compute-heavy process is starved of CPU along with an I/O-heavy process in their experiments in a microsecond-scale storage device. But the problem is not caused and is not specific to XRP. But XRP exacerbates the problem. [19]

2.5 Test Scenarios

BPF-KV is an author-designed simple key-value store to test the performance of XRP. It uses a B+-tree store and provides a method to create trees and operations. Using BPF-KV, they test the latency and throughput in "read" and range query scenarios. They also test tail latency when thread scaling. [5]

WiredTiger [11] is a real-word database an it is also a key-value store. In addition to migrating the above tests, they also experiment insert and scan operations.

Note that both in the BPF-KV and WiredTiger, the authors integrate the BPF function and modify the syscall `read_xrp`. In their experiment result, the overall performance of XRP is better than other baselines, like SPDK. Although the average lookup latency in BPF-KV is larger than SPDK, both of them are very faster and XRP avoids the SPDK's problems. Besides, WiredTiger shows that the XRP only benefits the lookup operation, other operations need I/O operations that cannot be benefited from the resubmission logic. As a result, these operations like scans cannot benefit a lot from XRP. [2] [19]

3 Artifact Evaluation

In the following section, we will explain how we managed to rebuild the original system and evaluate the result from the original paper.

3.1 Rebuilding the System

Using the script provided by the authors, we were able to build a patched version of the Linux kernel [4]. However, we encountered some obstacles during the building process.

- The provided kernel config is not suitable for certain situation. On Ubuntu 22.04 [9] the resulting kernel image did not have `CONFIG_BLK_DEV_NVME` enabled, which causes the system unable to detect the NVMe drive.
- The shell script has a lot of bugs which lead to permission errors. At some point the script creates a file with root priviledges, but tries to wirte to the file without privilege afterwards.

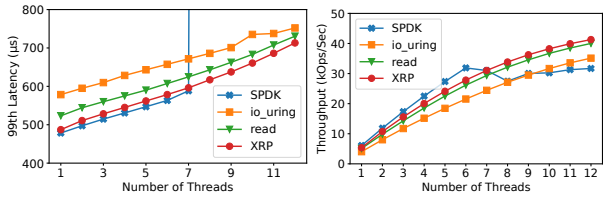

Figure 2: Testing Results

With aforementioned problems solved, we successfully built the patched kernel (5.12.0-xrp+) together with some utility programs in a virtual machine with the NVMe drive passed through using *vfio-pci* [10] for testing.

3.2 Evaluation Process and Analysis

The test was done in two parts, where *read_xrp* was compared to normal *read* syscall, *io_uring* [3] and *SPDK* [18] on top of two applications. BPF-KV [5] was used for demostrating the effectiveness of XRP, and WiredTiger [11] was used for a real world benchmark. We tested the performance of XRP in terms of latency and throughput under different workloads and situations.

Soon we found out that the performance of the syscall *read_xrp* was very unstable, both the latency and throughput varied a lot from each individual invocations. Through further experiments we determined that the problem lies in the CPU. Modern CPU throttling or multitasking technologies such as simultaneous multithreading (SMT) [15], turbo boost [14] and cpu frequency scaling [12] were affecting the stability of the performance.

After all these CPU technologies disabled, we finally managed to reproduce the experiments very consistently with stable results. To make sure the hardware is as similar as possible to the original authors', we created the environment as follows:

- patched kernel with Ubuntu [9] running in *QEMU-KVM* [13]
- 6-core CPU with SMT disabled, locked at 3.6 GHz, all performance governer disabled.
- 16 GiB of RAM
- Dell enterprise level PowerEdge series NVMe SSD passthrough using *vfio-pci* [10]

3.3 Evaluation Results

The testing result in Figure 2 shows that XRP delivered expected performance in our environment. It provides consistently lower latency as well as higher throughput than *read* and *io_uring*. *SPDK*, as expected, is faster than XRP under some conditions, but as soon as the number of threads surpasses physical CPU core count, its performance would be drastically degraded.

Despite the performance of XRP is consistently better than native Linux syscall, our result is still way slower (by order of magnitudes) than what was demostrated in the original paper [19]. With some experiments we conclude the reason why our experiment is so slow in the following points:

- **Hardware Difference** the NVMe drive used by the authors is Intel Optane P5800X, whose performance is way better than what we had. On paper, it delivers roughly twice the read speed, five times the write speed and twice the IOPS(Input/output operations per second). Other hardwares may also affect the performance but we believe the disk is the main factor.
- **Virtualization Overhead** our experiments were run inside a virtual machine, the authors on the other hand was doing theirs bare metal. A study suggested that using NVMe with vfio passthrough would add performance penalty in the neighbourhood of 20% [17]. Depending on the actual hardware, we believe this also affect our result in a similar manner.
- **Server Resource** our testing server was used by many people at the same time. Although the workload was not particularly high, this will definitely influence the results.

In conclusion, we managed to reproduce the result provided in the paper and we recken that the artifact provided by the authors is sufficient for other researchers to replicate the results reported in the paper.

Research and Implementation

4 Research Proposal

4.1 Continue the XRP Research Flow

After we have understood the research flow of the authors: They start from the existing problems and propose a new scheme XRP, using eBPF to accomplish bypassing most of the kernel layers. With some design principles and tricks, they have successfully solved the problems that other full kernel bypass methods have. At the same time, they give us a very good example to learn how can we use the advantages of eBPF to reduce the I/O time.

The first step is to start from a small point. Because the functionality of BPF-KV being relatively clear and easy to track, we first follow the BPF-KV [5] to figure out how can they construct the eBPF function to execute the corresponding functions, like range query.

The range query is a very common operation in data processing, which aims to return a series of an array that fulfills

our requirements. When we interact with data stored on disk in user space, we can use XRP to accelerate the query speed. We figure out the principle of how can it return target key-value pairs. In this process, we find that the normal way is to use *pread()* [2], and every time it needs to traverse the whole kernel stack. And authors only provide a few operations in the BPF-KV. So can we add more operation functions like aggregation functions to test the performance between the normal mode and XRP?

In the second step, we go deep into the kernel to see detailed implementations. If we use XRP, what is the flow of the process? We go with the flow from the user space and use *syscall()* to enter the kernel. In the kernel, we check the author-defined function *read_xrp* [4] and corresponding resubmission logic. We are inspired by B+-tree store and want to expand XRP to some other similar data structures that can be used in the XRP. Another structure that ensures this functionality is the ext4 file system.

During check the codes, we find authors define a resubmission counter [4], but they don't use it to limit unbounded execution. And according to the XRP current serially submit the command, we propose to change it into submitting in parallel.

When we check all of the implementations of XRP in the user space and in the kernel, in the last step, we treat the project [6] from the Linux architecture angle. We notice the current XRP only supports the 5.12 kernel version [4], so we try to port the whole project into a newer Linux version. Besides, as we mentioned in the chapter 2.4, the current process scheduler is not "fair" for the computation-heavy task and results in a starvation problem. So we want to improve the scheduler.

4.2 Proposals Analysis

After a complete analysis of the project, we propose our research improvements from the BPF-KV, XRP implementation, and system architecture level. We will discuss them in detail in Chapter 5.

With the improvement of BPF-KV, we can expand the test and application scenarios of XRP so that we can check the performance in more test scenarios and improve the application ability of XRP. By changing some implementation details and porting XRP to a newer version, we will improve the performance based on the authors' contribution. Also, we could use newer Linux features and simplify the whole structure. Lastly, we can experiment with different approaches to avoid the starvation problem.

However, we still have faced some technical challenges to solve these research proposals. Like in other data structures, situations are not identical for us to fetch the next node's physical address. And it is hard for us to simulate the starvation scenario in the paper. [19] Next part we will focus on how we have solved these challenges.

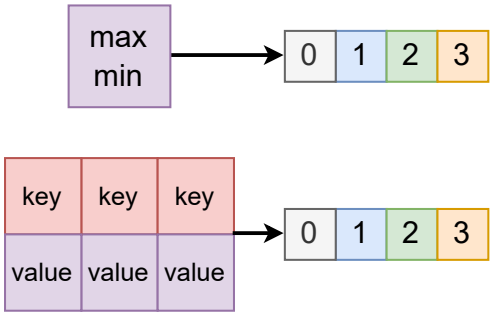

Figure 3: **Aggregation operations**: the upper one introduce a new variable to operate over single elements, the other one introduce a key-value pair array to operate aggregation over multiple elements

5 Design and Implementation

To achieve the goals in the research proposal, we have carried out a series of designs on XRP. A gradual deepening of understanding of XRP accompanies their implementation.

5.1 Aggregation Operations

The original XRP approach implemented support for B+ trees and Modified WiredTiger. But they focus on queries on a single value. So we add the aggregation operations here, the user specifies the size of the interval to be searched, and the program randomly initializes the starting index to calculate the actual search interval.

Our specific operations are divided into two types, which are shown in Figure 3: (1) accessing multiple elements and then returning one value, and (2) facilitating multiple elements to return multiple values. The first is by adding an extra variable to enable the operation on newly traversed elements. The second needs modifying the structure of the query struct to support continuous queries and the returns which contain multiple pieces of data. Besides, we designed two kinds of return format. The first kind is to return an array of the query range. The second kind is to return a set by checking and deleting the duplicate elements and returning elements that appear once.

5.1.1 Range Operation on Single Element

In this part, we implemented max, min, sum, and avg operations. For the max and min operations, we maintain a newly set variable, and each time the elements are traversed in the set interval, they will be compared with this new variable, and finally, the corresponding result will be returned. For the sum and avg operations, their implementations are relatively close. First, we maintain a totalSum variable, and then we use the second counter variable. In this way, after the traversal is over,

the desired value can be obtained through the calculation of the two variables.

The simpleKV uses char arrays to store the value of each node, but the final result we want is a specific number. So we also need to convert the char array to a concrete number. In the general case, for the user mode, the Stolen function can be easily applied to solve this problem. But for programs accelerated by eBPF in kernel mode, functions in the standard library cannot be used directly. So we manually implemented the corresponding function tool for calculation.

5.1.2 Range Operation over Multiple Elements

In this step, we set the query structure to contain multiple elements. To be more specific, we introduce a key-value pair array. Intuitively, an index variable is added to the query struct to visit the targeted element accordingly. It should be noted that in the logic of eBPF, the query struct occupies a pre-allocated continuous space. This space is pre-allocated to the struct by the user mode, and then the first address passed in is parsed in a specific way same in the kernel. After operating this alignment, the in-kernel function and userspace code can have consistent logic.

After adjusting the framework including the query struct, the rest of the work is to add the corresponding process in the kernel query function: if an element is found, we put the element into the position where the index of the key-value pair array is the *index* and make the index increase. When the userspace main program starts to receive the return value, it passes out the whole key-value pair array. The corresponding elements can be read and finally returns an array or a set according to the user's choice.

In the actual implementation, the continuous space of the query struct needs to be pre-specified. We set this value as 0x1000. In this case, the maximum number of elements that can be queried and stored in each query struct is also limited by the given space. Considering all other variables needed in the query struct to maintain the query, the final number of elements that could be fetched from once query is 32.

5.2 Apply More Data Structures

After optimizing the original XRP querying processing, we think about whether eBPF can be extended to support more data structures. In this part, we designed two data structures: easyArray and SkipLlist.

5.2.1 EasyArray

EasyArray is a simple array, the main structure and logic are shown in Figure 4. The core idea is to design an intensive disk-io task. Using a high-frequency reading task to verify the effectiveness of the XRP method. EasyArray is a contiguous sequence of address spaces distributed with randomly generated elements. In this array, the value of the corresponding

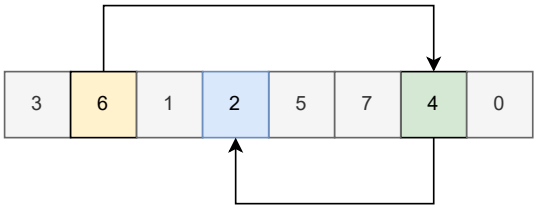

Figure 4: **Querying Process of easyArray**: Two continue jumps are shown in this figure.

position can be obtained directly through the index. The tasks performed by easyArray will first be given an initial query index by the user, and then the initial value of the corresponding position can be obtained. Then the value is performed in a specified operation, the result will be used as the new query index. The above process is named as a jump. When the number of jumps specified by the user is completed, the final value is returned. The main logic is shown as the following code:

```
value     = easyArray[index];
new_index = op(value)% easyArray.length;
```

The selection principle of the $op()$ function is that the simpler the better because the calculation of numbers will not be accelerated in the kernel. And the simpler function calculation still jumps the address and reads on the hard disk, which leads to the same realization result. The core purpose of designing easyArray is to compare the execution speed of the same data structure and logic in user mode and eBPF. Simple calculations will increase the proportion of hard disk operation tasks, thereby enhancing the acceleration effect of the XRP method on this task.

In the actual implementation, easyArray stores 1000 consecutive elements, and the type of the stored elements is int. The first position of the query is specified by the user, and the number of jumps is set for each query task. The value obtained from the previous query is calculated by $op(value) = 2 * value$, and the modulo is performed on the total length to obtain an effective and safe new index.

5.2.2 SkipList

The introduction of SkipList is to verify whether XRP logic can support complex data structures. So the standard form of skiplist is used directly, the data structure is like Figure 6.

After completing the data structure design and operating logic, it needs to be implemented in the corresponding eBPF program. The main difficulty in the implementation process is that the kernel mode can only be implemented using the features of the basic C language, and it needs to meet the requirements of the eBPF verifier in terms of security.

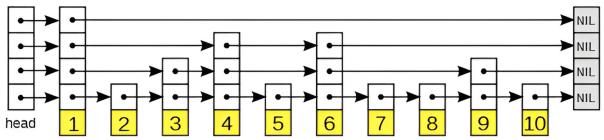

Figure 5: SkipList Data Structure

5.3 Limit XRP Exectution Times

After in-depth research on the implementation of XRP, we found that the original paper did not restrict the execution times of XRP. In the Linux kernel, the main scheduling function of XRP is called in the nvme driver. The core logic is abstracted into the form of Listing 5.2.1.

```
void nvme_handle_cqe(){
    xrp_process();
    if (finished){
        fetch_result(&result);
        return;
    }
    nvme_submit_cmd(new_request);
}
```

It will detect whether the execution of the driver function is complete, if the execution is complete, it will save the corresponding state and then exit; if it is not complete, it will trigger the function again and enter a new query process. Based on this logic, if meeting a kernel read function with many levels of execution, such as easyArray that sets 10,000 iterations, the query subtasks created by the eBPF query process will occupy a large proportion of the whole nvme driver execution tasks. This will reduce the proportion of other tasks, and may even starve other tasks.

To make up for this shortcoming, we add a counter limit to XRP, implemented by an atomic counter. The execution times of the in-kernel storage function are counted in the actual implementation. When each kernel function is executed, the atomic counter will be incremented. After a certain number of times, it will reach the boundaries, and the execution will be stopped. The core logic is shown in the Listing 5.3. At the same time, in the case of the early return, the temporary result of the current calculation will be returned with an additional flag variable to mark that the return is a temporary result.

```
// atomic counter increment
atomic_long_inc(&xrp_ebpf_count);
if(xrp_ebpf_counter.counter > MAX_XRP_COUNT){
    bool interrupt = true;
    fetch_result(&result, interrupt);
    return;
}
```

5.4 XRP-Parallel — Parallel XRP

In the XRP implementation, the system call and implementation are serial. An XRP task triggers the execution of another task after execution. A natural idea is to adjust the serial task scheduling to parallel scheduling so that better performance can be achieved.

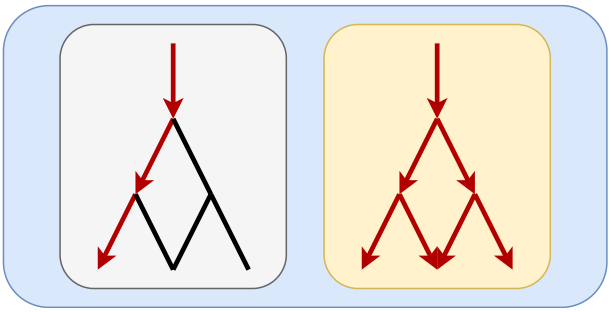

Figure 6: **Original XRP(left), XRP-Parallel(right)**: XRP-Parallel parallel the execution of XRP, target achieving better performance

XRP-Parallel is designed based on the above ideas. It can submit requests to two or more branches at the same time after the execution of a request is completed, which has achieved the purpose of parallel execution. Following the intuitive design idea, we define the parallel request structure and request queue.

If we just submit them one by one through the queue, it is not truly parallel. So we design to send the generated tasks to different NVME execution queues for parallel execution. Because we are modifying the nvme driver in the Linux kernel, this design is feasible.

In the final implementation, limited by the understanding of the nvme driver and the great difficulty of debugging in the Linux kernel, we implemented the corresponding interface and did not compile the latest version of the kernel.

5.5 Linux Kernel Porting

The Linux kernel [8] is developing rapidly. Since XRP was created, a lot of new features and functions were added to the kernel [7]. In order to benefit from the newly added functionalities in terms of performance and ease of development, we targeted to port the XRP patch to the latest longterm support version of the Linux kernel.

We planed to port the patch to Linux 6.1 lts. However, the block device and *io_uring* part of the kernel received some significant changes in older versions of the kernel. As a result, some of the functions used by XRP got removed and the replacement function requires more information than before. Due to the lack of knowledge of how the internal of Linux kernel works, we decided to take a step backwards and target Linux 5.15 lts instead.

Porting to 5.15 was successful. We managed also to make use of some new kernel functions to simplify as well as improve the logic for XRP. A non-exhausting list would include:

- Submitting a batch of *nvme_request* at once instead of submiting sequentially using `nvme_submit_cmds`. This improves the performance as it allows a batch of command share a same *spinlock* to avoid lock and unlock for each individual request.

- Changed logic for initializing *nvme_iod* struct. The updated version of this struct includes a field for *nvme_command*, this makes it possible for XRP to save a `kmalloc()` call for every single request.

- Setting up *io_uring* using updated *io_op_def* struct.

- Removed some redundant performance counters. Some of the unused atomic variables were left in the codebase. They were presumably used for developing purposes but was not cleaned up.

In conclusion, we ported the kernel patch to a newer long term support version, and improved some aspects of the XRP logic.

5.6 CPU Scheduler

As mentioned earlier, XRP aggravates the starvation of CPU based tasks in CFS (Completely Fair Scheduler). In order to allocate more CPU times to XRP we targeted finding a better scheduler for XRP.

We originally planned to experiment with different kernel schedulers such as BFS, MuQSS and then design load balancer with the help of *sched_ext* class. However, we cannot replicate the starvation in our machines. With some experiments we came to a conclusion that the context switch in our machine is less frequent than Optane SSD because our hardware is 5 times slower in read speeds. As we lack the knowledge the way to compensate the hardware difference in a program, we decided to target an loadbalancer with cpu-pinning. [16]

The idea is to reserve most of the CPUs for CPU-based threads by cpu-pinning. For example in a VM system with 8 cores. 6 cores are pinned exclusively to cpu-heavy tasks. The pinning process is as follows:

- Step 1 eBPF programs using *sched_setaffinity* are hooked to kprobe

- Step 2 the programs pin the threads to vCPU with *sched_setaffinity* system call. In this step most vCPUs are *reserved* to CPU-heavy tasks.

- Step 3 kprobes propagate the systemcalls to a QEMU hypervisor each invocation.

- Step 4 vCPUs are mapped to pCPUs.

6 Results

Due to a severe hardware failure happened at the end of the project phase, we were unable to design a meaningful test for our own implementation on the server. The performance of the NVMe drive degraded drastically in the virtual machine even without any modification from our side. Our experiment showed that it could only deliver up to $\frac{1}{5000}$ of the original speed in terms of raw performance. After some experiments and journal analysis as well as crashing the physical server several times, we believe that that the vfio driver configuration on the server is defected. The performance degradation only happens specifically in the virtual machine with vfio [10] passthrough, for both normal *read* and *read_xrp* syscalls. However at this point the project phase has already ended and we didn't manage to fix the server on time.

Therefore, all the tests we ran for our own implementation were done in an unstable environment with extremely low performance, and the result is not reproducible or stable. The throughput varied from 0.1 IOPS to 50 IOPS each time, and the latency faced similar situation. With such low performance any improvement to the original artifact cannot even be observed.

It's quite unfortunate that we cannot test our own code in a real working environment, which in theory would improve the performance of XRP. But even with such performance, we were still able to observe that *read_xrp* is under most circumstances faster than *read*.

7 Conclusion

In this project we verified the effectiveness of XRP and ported its patch to a newer version of the Linux kernel. We simplified and improved its logic in the kernel space. We optimized BPF-KV, implemented new data structures and functionalities. Although the hardware issue prevented us from progressing, we were still able to implement a example as proof of concept.

Availability

Our contribution including modified Linux kernel and BPF-KV can be found on GitHub [6].

References

[1] eBPF. https://ebpf.io/.

[2] pread(). https://man7.org/linux/man-pages/man2/pread.2.html.

[3] Ringing in a new asynchronous I/O api, 2019. https://lwn.net/Articles/776703/.

[4] GitHub organization for xrp-project, 2022. https://github.com/xrp-project.

[5] GitHub page for BPF-KV, 2022. https://github.com/xrp-project/BPF-KV.

[6] GitHub organization for sys-lab-xrp, 2023. https://github.com/ws-22-sys-lab-xrp.

[7] KernelNewbies: Linux changes, 2023. https://kernelnewbies.org/LinuxChanges.

[8] The linux kernel, 2023. https://www.kernel.org.

[9] Ubuntu - enterprise open source and linux, 2023. https://ubuntu.com/.

[10] VFIO - virtual function I/O, 2023. https://docs.kernel.org/driver-api/vfio.html.

[11] Wiredtiger storage engine, 2023. https://www.mongodb.com/docs/manual/core/wiredtiger/.

[12] Ionut Anghel, Tudor Cioara, Ioan Salomie, Georgiana Copil, Daniel Moldovan, and Cristina Pop. Dynamic frequency scaling algorithms for improving the cpu's energy efficiency. In *2011 IEEE 7th International Conference on Intelligent Computer Communication and Processing*, pages 485–491, 2011.

[13] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46. USENIX, 2005.

[14] James Charles, Preet Jassi, Narayan S Ananth, Abbas Sadat, and Alexandra Fedorova. Evaluation of the intel® core™ i7 turbo boost feature. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 188–197, 2009.

[15] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading:a platform for next-generation processor. 1997.

[16] Leonardi L., Lettieri G., and Pellicci G. ebpf-based extensible paravirtualization. In *International Conference on High Performance Computing*, page 383–393. Springer, January 2023.

[17] Ben Walker. High performance nvme virtualization with spdk and vfio-user. 2021.

[18] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, 2017.

[19] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.