Bachelor Thesis

Institute of Computer Science, FU Berlin & Televic Rail GmbH

Performance-Centric Device Access: Evaluating Kernel I/O Stack Bypass Methods in Embedded Systems

Jacob Elias Thiessen
Freie Universität Berlin
Mt.Nr.: 5224517
jacob.thiessen@fu-berlin.de
+49 177 6364 175

supervised by: Dr.-Ing. Barry Linnert
Freie Universität Berlin
linnert@inf.fu-berlin.de

Contents

1 Introduction **4**

2 Background and Related Work **8**

2.1 Traditional Systemcall-Based I/O: A Primer . . . 8

2.1.1 Requesting Action . . . 9

2.1.2 Contex Switches . . . 10

2.1.3 Memory Mapping . . . 10

2.1.4 Drivers . . . 11

2.1.5 Kernel Return Mechanism and Interrupts . . . 12

2.2 Rethinking Systemcall Overhead: A Gateway to Optimizations . . . 12

3 Evaluating Kernel I/O Stack Optimization Techniques **13**

3.1 io_uring . . . 15

3.2 eBPF . . . 19

3.3 DPDK & SPDK . . . 21

3.4 vFIO . . . 23

3.5 Common Optimization Methods . . . 25

4 Benchmarking and Results **26**

4.1 Experimental Setup . . . 26

4.2 Results . . . 27

4.3 Analysis and Discussion . . . 31

5 Conclusion **33**

1 Introduction

Modern computers have come far since the days of punch cards or the hand woven code of the Apollo Guidance Computer[11]. The computer scientists of old worked on *bare metal* machines, without the cushy security of operating systems, high level abstracted programming languages or the calming presence of clippy to keep them company. One of the most important rungs on the ladder to the broad adoption of user electronics was the introduction of the kernel. The concept was first introduced in the time sharing operating systems for the MIT computing cluster, initially released in 1961 for the IBM 709[28], a computer so comically large it weighed more than the Daimler-Benz car "Smart" (see Figure 1). The kernel / user space dichotomy has since come to be a staple of any modern operating system with any relevant measure of complexity and features on every home computer, phone, or smart device, facilitating a peaceful coexistence between flocks of heterogeneous consumer software products with heterodox concepts of "polite behavior" in shared execution environments

Figure 1: IBM 709 Data Processing System (ca. 1961)[9]

Since the kernel has to be kept uncoerced by the machinations of other processes, modern computing systems are bifurcated into two distinct domains. User space, also known as *userland*, references the isolated space that user facing programs spend their lifetime in. Programs that exist in user space operate in an isolated, fault tolerant, and permission checked environment provided by the operating system, where binaries operate without their procedures endangering system integrity. Their operations are limited to their own, corded off, memory spaces and their execution time is limited by the scheduler, which distributes a fair balance of computing time among competing programs running on the computer. Kernel space on the other hand, refers to the space exclusively reserved for the kernel. Here resides the functionality that is sensitive to misuse or fault. The kernel presides over the aforementioned scheduling, it aggregates a collection of drivers and controls the memory mapping and permission checks that enable the encapsulation between user programs[8, p.36].

This approach to operating system design is called **monolithic** and is the prevailing design of the Linux kernel. Other architectures such as microkernels or exokernels offer a more *laissez-faire* attitude to hardware access, trading the full abstraction of monolithic systems for performant, user space managed hardware with only minimal remnants of privileged kernel mediation, thereby granting applications finer control at the cost of increased complexity and potential security concerns[23, 29].

The strict separation between *userspace* and *kernelspace* is enforced at the hardware level, integrating checks on execution environments into the CPU itself. Most modern architectures, such as x86, implement multiple hierarchical privilege levels, with user space operating at a lower privilege level and the kernel operating at the highest. This architectural design ensures that user programs cannot directly manipulate sensitive hardware or kernel functions without explicit permission, forming the basis for secure and stable computing. ARM, the hardware architecture of choice for low power embedded systems such as the Raspberry Pi, uses a different terminology (*supervisor mode / user mode*), reducing the amount of hierarchical increments in privilege to two[39](Figure 2), but still maintains the fundamental separation between user and kernel modes, ensuring secure and controlled access to system resources.

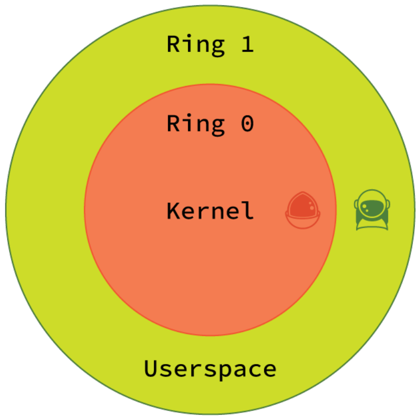


Figure 2: 2-Tiered protectiong ring architecture

For many features user space programs require access to functionality which lies exclusively in the purview of the kernel, since it controls access to the file system, the networking peripherals, and the input devices. To facilitate the interaction between the critical components of the computer and the programs in need of their functionality, the kernel exposes an Application Programming Interface (API) to request actions, referred to as system calls.

> ”At a high level system calls are ”services” offered by the kernel to user applications and they resemble library APIs in that they are described as a function call with a name, parameters, and return value.” - Linux Kernel Community[12]

System calls have traditionally been implemented using **synchronous and blocking** functions that halt the operation of the calling program until the function completes and returns [12]. There is an inherent simplicity to the control flow enabled by this methodology as it allows programmers to use the syntax of assignment without having to concern themselves with whether a resource is currently busy, an awaited packet is a couple of milliseconds late or the usb stick that contained the file that was being written to was pulled halfway through the operation. System calls enable an easy to understand, high level method to abstract from hardware but it puts system calls in the critical path for any program[36, 24].

Since system calls are integral to the execution of user-level programs in operating system build on monolithic kernels, their performance becomes a critical bottleneck. *Unfortunately system calls are slow.* This means that, when interacting with sluggish or saturated devices, blocking system calls can cause significant delays in process execution [33]. These inefficiencies are not merely a matter of delayed task completion; successive waiting time stacking by multiple programs prevents systems from fully utilizing available hardware performance or can lead to process starvation.

That poses the question: *What makes system calls inherently slow operations?* The answer lies in the user space/kernel dichotomy itself. Executing a system call necessitates a context switch from user mode to kernel mode, which involves which involves switching the processor to a higher privilege level, saving and restoring CPU state, and incurring overhead from security checks[33].

Although system calls are the standard interface for modern personal computers, the criteria on security and fairness are somewhat loosened in the context of embedded systems, servers or High-Performance Computing (HPC) computers. Since such computers are not designed to be accessed by the typical Dumbest Assumable User (DAU), they often operate within a more relaxed security environment. Software on such systems is vetted before the system is deployed, resource needs are known *a priori* and tested against and remote access is highly limited. This opens the door to a more performance centric view on system calls. Over the years, researchers have developed a wide array of techniques aimed at circumventing the performance overhead typically associated with the traditional system call interface, seeking more efficient ways to interact with the kernel and the hardware components it controls, improving overall system performance. These advances are sharply pulled into focus when discussing the growing range of embedded Internet of Things (IoT) products, where the need for streamlined interfaces becomes critical to ensure predictable, low-latency behavior in systems designed for edge computing workloads[48].

In this thesis, we explore how the fundamental separation between user space and kernel space, while critical for security and stability, introduces notable performance limitations. We focus on embedded systems, where relaxed security constraints and predictable application profiles open the door for performance-centric optimizations. This work investigates alternative mechanisms for interacting with hardware that bypass the traditional system call interface, aiming to reduce latency and improve throughput.

Televic Rail

This thesis was developed in cooperation with **Televic Rail**, a company specializing in embedded computer systems and software solutions for public transportation rail stock, including passenger information systems, fleet management, and cabin surveillance. Televics systems are deployed within the I/O-heavy context of train cars. Interactions between information displays, ticketing machines, passenger counting devices, and surveillance cameras builds the core of much of Televics software products and is therefor of immediate interest when deliberating on potential performance increases.

Recent development efforts have migrated the Televics products from platforms utilizing Windows Embedded to the Linux operating system Debian, marking a historical shift for the companies design philosophy. Moving into a new era of decade-appropriate operating systems for embedded devices, a plethora of interesting possibilities have been made available when inspecting how our proprietary hardware components are addressed by our software solutions such as the Passenger Information System (PIS). One of those components is the Multifunction Vehicle Bus (MVB) card, a device facilitating access to a part of the Train Communication Network (TCN) connecting devices within a single cabin via a shared data bus. As the number of devices integrated into train cars rises with the introduction of modern functionality like , communication with the hardware that abstracts communications over the bus toward the software solutions Televic sells, becomes busier. In anticipation of an ever-growing number of sensors and endpoints, providing a growing range of services to the passengers aboard trains utilizing Televics broad range of systems, the MVB card will serve as a stand-in throughout this thesis, to explore the interaction between hardware drivers and user space applications in optimizing system performance and communication efficiency.

The research conducted in this thesis aims to directly support Televics objective to deliver performant and responsive embedded systems based on the Linux operating system. Evaluating alternative approaches to kernel I/O stack interaction provides valuable insights for ongoing and future product development.

2 Background and Related Work

In this section, we will explore the system call interface and its mechanisms in detail, while noting that a full exploration of the underpinnings of the Linux kernel is beyond the scope of this work. We aim to build a mental model for the reader that will allow an understanding of how system calls can introduce unproportional overhead in specific use cases and how the later introduced optimizations techniques and their approaches alleviate these. Also this primer will serve to further understanding of just how reliant on kernel action user space programs are, and how much insight can be gained by evaluating a programs use of system calls (see Figure 3).

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 27.50    0.000303          37         8           mmap
 18.15    0.000200          50         4           munmap
 14.79    0.000163          40         4           mprotect
  8.71    0.000096          16         6           close
  7.26    0.000080          20         4           openat
  6.17    0.000068          13         5           newfstatat
  4.45    0.000049          16         3           brk
  2.18    0.000024          24         1           getrandom
  2.09    0.000023          11         2           read
  2.09    0.000023          23         1           prlimit64
  1.72    0.000019          19         1           set_tid_address
  1.72    0.000019          19         1           fadvise64
  1.63    0.000018          18         1           rseq
  1.54    0.000017          17         1           set_robust_list
  0.00    0.000000           0         1         1 faccessat
  0.00    0.000000           0         1           execve
------ ----------- ----------- --------- --------- ----------------
100.00    0.001102          25        44         1 total
```

Figure 3: Complete `strace` output of a `cat` command on an empty text file, showing all invoked system calls.

2.1 Traditional Systemcall-Based I/O: A Primer

As noted in the introduction, system calls are services exposed by the kernel analogous in role to how libraries like pythons mathematics library `NumPy` or `React`, a library offering an ecosystem for UI development, offer importable functionality to be used within other code. In contrast to functions imported from libraries, system calls do not reside in user space; they are implemented via hardware dependent CPU instructions[52], that transition control into the kernel. To understand the underpinnings of this procedure we will follow a simple `write()` call as implemented by the c programming language's `unistd.h` interface, to trace its path from user space into the kernel and back. Along the way we will touch upon how user space programs request action through the system call interface, how context switches and mode switches work, how drivers are addressed and function, and how memory mapping works to isolate processes from one another.

2.1.1 Requesting Action

As an entry point into understanding system calls, we examine the execution path of the `write()` function. The following program demonstrates a minimal invocation of `write()`, saving a string to an existing text file.

```c
int main() {
    const char *text = "Hello, world!\n";

    int fd = open("myfile.txt", O_WRONLY);
    write(fd, text, strlen(text));
    close(fd);

    return 0;
}
```

Figure 4: Minimal C program demonstrating a direct system call-based write of a string to a file.

At first glance `write()` *writes* given content to a file, but Linux obstructs many complex endpoints behind the UNiplexed Information Computing System (UNIX) mantra ”everything is a file”, treating devices, sockets, pipes, and even some hardware interfaces as files. This means that, although in this example, `write` is used to append content to file, the invocation `write(1, "Hello, world!\n", 14);`, would output the string to the terminal via. `stdout`, the default output stream[49]. To achieve this form of polymorphism, the `write` function wraps the GNU C Library (glibc) which in turn provides implementations for the write function interface as defined in the Portable Operating System Interface (POSIX), an interface enabling the interoperability between UNIX-like operating systems. The call ultimately invokes the generic `syscall()` function under the hood whose signature takes an input parameter that indexes into the system call table, a list of available functionalities offered by the kernel[55, 1, 8] (see Figure 5). This cascade allows for the call to arrive at the endpoint appropriate implementation, be it file system, Pseudo Terminal (PTY) or `stdout`.

`long syscall(long number /* <- the aforementioned index */, ...);` [31]

```
0 common  read      sys_read
1 common  write     sys_write
2 common  open      sys_open
# <- the aforementioned index
3 common  close     sys_close
4 common  stat      sys_newstat
5 common  fstat     sys_newfstat
6 common  lstat     sys_newlstat
7 common  poll      sys_poll
8 common  lseek     sys_lseek
9 common  mmap      sys_mmap
10  common  mprotect    sys_mprotect
```

Figure 5: Extract from the syscall_64 table

2.1.2 Contex Switches

Within Linux, processes possess their own contexts, consisting of a private virtual memory and their execution state, represented in the content of the relevant Central Processing Unit (CPU) registers. Context switching between processes involves storing execution state of the running process and restoring the running state of another. The low level details of how this swap is achieved are subject to variance of specific hardware but generally don't come for free as registers have to be saved to memory, stack and execution pointers stored and restored, and virtual memory context changed. Estimates of the cost of context switching on modern standard CPUs range from 1.2 to 1.5ms [47, 5, 33]. When a process performs a `write` system call, it triggers a switch from user mode to kernel mode, ensuring the process has the necessary privileges to modify data. The switch to kernel mode, known as mode switch, is less costly than switching between `userland` applications since the memory context remains unchanged. However, it still requires flushing the CPU pipeline and performing privilege checks[8, p.102].

2.1.3 Memory Mapping

Every process in Linux exists in an isolated context, granting the illusion of executing in its own memory space, beginning at address `0x0`. This allows software developers to neglect the difficult task of address traversal and obfuscates the physical memory. The addresses accesses by the process are called logical address. When accessing this address, the operating systems Memory Management Unit (MMU) translates this address into a physical address, referencing actual hardware locations. The mapping of these addresses is maintained by so called page tables. Each process possesses a Page Table Entry (PTE), aggregating a memory space that seems contiguous to the user space program but might reference a disjointed collection of memory locations on the hardware. These tables are hiearachical, cascadingly mapping each PTE to the next table, until the final entry resolves to a physical address. Since keeping all those translations quickly accessible all the time is unfeasible for systems with many programs the MMU keeps a special cache of recently applied translations called the Translation Lookaside Buffer (TLB), in an attempt to shortcut translations efforts. When the TLB contains the desired address, this is referenced as a *cache hit*, if not, a *cache miss*[13][8, p.35].

To optimize memory management, modern systems employ the concept of huge pages, which are larger memory pages compared to the standard page size. Huge pages reduce the overhead of managing large amounts of memory, as they require fewer entries in the page tables, leading to improved TLB efficiency. They are particularly beneficial in applications requiring large memory allocations.

When `write` is called from user space, the address to which data will be written will therefore be resolved using the process's memory mapping, translating the logical address into a physical address through the page tables.

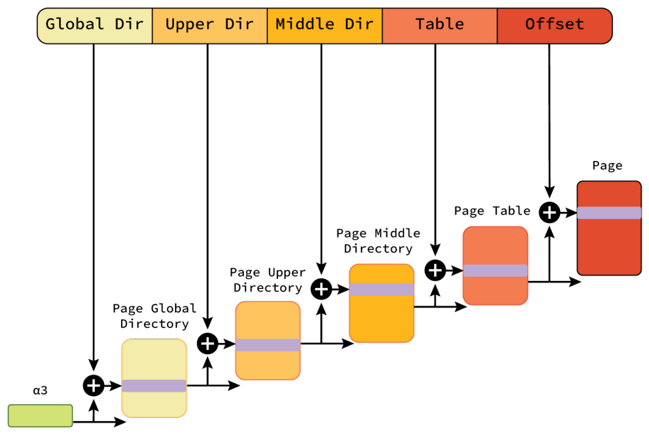


Figure 6: Illustration of Linux paging cascade from logical to physical address

2.1.4 Drivers

Drivers are software that facilitate access to hardware components or similar low-level endpoints such as the Virtual File System (vFS), a Linux-specific abstraction that provides a uniform interface for interacting with different file systems, hiding the underlying complexities of storage devices. The `syscall()` function's abstraction into the system call table points toward the `sys_write()` systemcall. This function is function gate into the relevant driver. When writing a driver for the linux kernel, one of the key objects is the file operations struct (fops)(see Figure 7). This is where the actual implementation of operations like `read`, `write` and `open` reside. If we take a look at the implementation for writing to a file, the `sys_write()` call eventually invokes the .write function pointer defined in the corresponding fops structure[8, p. 540][46]. For regular text files, this pointer leads into the vFS layer, which in turn dispatches to the concrete filesystem driver implementation, in our case, ext4, the filesystem of choice for Raspberry Pi systems. The actual forwarding into the driver code is actually done by a subsystem called the system call dispatcher, which transacts the mapping between the system call and the appropriate driver.

```
struct file_operations my_fops = {
 llseek:  my_llseek,
 read:  my_read,
 write:  my_write,
 ioctl:  my_ioctl,
 open:  my_open,
 release: my_release,
};
```

Figure 7: A partial file operations struct

2.1.5 Kernel Return Mechanism and Interrupts

In Linux, when a system call completes, the kernel passes return values back to the user-space program via CPU registers. The return value typically indicates the success or failure of the operation, with specific error codes returned in case of failure. For example, in a write system call, the kernel will return the number of bytes written or an error code if the operation fails. When the kernel passes a task on to a hardware component, such as a network card, it will resume other tasks until the component is finished. The mechanism by which the kernel is made aware of the need to interact is called an interrupt. These interrupts temporarily halt the current execution to handle the event, after which the kernel resumes normal operation, often passing the result back to the calling process. On completion of the `write` call, the kernel is interrupted by the file system and passed the number of written bytes, passing them back up to the calling program. This mechanism ensures that the kernel can efficiently handle asynchronous events without constantly polling for results[35][8, p.131].

2.2 Rethinking Systemcall Overhead: A Gateway to Optimizations

As modern hardware like Non-Volatile Memory Express (NVMe) Solid State Drives (SSDs) push read and writing speeds, the proportion of the small cost of an operation taken up by the overhead introduced by the mode switch becomes larger. On modern architectures it has been estimated to make up 50% of the execution time for memory access[57, 58]. This growing overhead presents a strong case for optimizing the kernel I/O stack to reduce inefficiencies. In the following section, we will explore various optimization techniques that address these challenges. By evaluating strategies like kernel-side polling and memory mapping, we aim to demonstrate how these techniques can enhance performance, particularly in environments demanding high throughput and low latency, and how they can be integrated into existing systems with minimal disruption.

3 Evaluating Kernel I/O Stack Optimization Techniques

This section presents a detailed evaluation of kernel I/O stack optimization techniques, with a focus on employed methodology, targeted performance layers, and ease of integration with existing software. This discussion aims to highlight commonly deployed design patterns such as kernel-side polling and memory mapping as core implementation strategies and will discuss how these approaches can be benchmarked against each other in a generalized way.

The Number of different methodologies for optimizing the I/O stack are quite large due to the heterogeneous nature of I/O. Where some techniques, such as io_uring were developed with a generalizable aproach, offering an performance increase for a host of use cases that utilize the I/O stack, frameworks such as epoll originated from projects with very narrow focus. In the following table we give a brief overview of a subset of relevant techniques, their application domains and typical use cases.

Technique	**Domain**	**Bypasses**	**Typical Use Case**
eBPF	Networking / Syscall Layer	Netfilter, tc, parts of kernel	Programmable packet filtering, syscall hooks, observability
io_uring	General Async I/O	Traditional syscall interface	Efficient async file and socket I/O with low syscall overhead
VFIO	Device I/O (PCIe, etc.)	Kernel driver stack	Secure userspace access to devices (e.g., MVB card, GPUs)
vDPA	Virtualized I/O	Emulated I/O device interface	Kernel-assisted virtual device passthrough, often with VFIO
DPDK	Networking / Storage (NVMe)	Kernel networking stack, block layer	High-performance polling-based packet and storage I/O (e.g., NVMe)
SPDK	Storage (NVMe, SSD)	Filesystem, block layer	Polling-based high-performance storage stack, optimized for NVMe
XDP	Networking (Socket Layer)	BSD Socket API, kernel net stack	Fast packet I/O with socket compatibility, zero-copy
epoll	I/O Multiplexing	select, poll	Efficient event notification for scalable network applications
UIO	Device I/O (General)	Kernel driver stack	Userspace drivers for devices, enabling direct interaction with hardware
fastcall	Systemcall Optimization	Traditional syscall interface	Reduces overhead in function calls by using a direct calling convention
MMIO	Device I/O (Memory Access)	Kernel device driver stack	Direct access to hardware memory, used for communication with devices like network cards or GPUs
AIO	General I/O	Blocking I/O system calls	Allows non-blocking I/O operations, improving throughput by enabling parallel operations

Table 1: Techniques for Kernel I/O Optimization

We will pick four of these techniques, covering a broad range of applied methodologies and use cases, to evaluate in depth. However, it is important to note at this junction, that the scope of this work will not permit us to do justice to all these variations. In this section, we will evaluate Extended Berkeley Packet Filter (eBPF), Virtual Function IO (vFIO), Data Plane Development Kit (DPDK) as well as its sibling Storage Performance Development Kit (SPDK), and the asynchronous interface io_uring, examining how and if these technologies can be applied to our example use case: the optimization of the MVB card, covering key representative techniques.

3.1 io_uring

Merged to the linux kernel on March 8th 2019 for v5.1-rc1, the asynchronous I/O interface io_uring was introduced to the kernel by Jens Axboe, a software engineer at Facebook, for the purpose of offering a generalizable, asynchronous method by which to address kernel level device access[3]. Io_uring builds on the limitations of the existing Linux-native asynchronous I/O library (libaio) by offering a more flexible and performant approach to user-kernel interaction, introducing memory-mapped submission and completion queues, batching and request polling.

io_uring achieves its asynchronous nature by offering two ring buffers that function as queues for requested and completed actions. Aptly named `io_uring_sq` *submission queue* and `io_uring_cq` *completion queue*, these two buffers offer a memory mapped interface for submitting and receiving I/O requests. From user space, processes interacting with io_uring enabled drivers, submit requests for kernel action by adding `sqe` (*submission queu entry*), structs that describe operations such as `read, write, ioctl` etc, into the submission queue and calling the system call `io_uring_enter` to notify the kernel of pending events. The program can then wait for a corresponding entry in the completion queue[27, 3, 4]. This sequence removes the necessity for the calling process to block during the time the request take to be handled, making user space-kernel interaction lockless. A boon on its own, io_uring is capable of more than asynchronous I/O context. Since the two queues can be mapped into user space, writing to these data structures does not require a mode switch. io_uring is also capable of receiving vectorized input, allowing for the userland process to batch-request action from the kernel without the need for an iterative interaction[14]. Furthermore io_uring allows for the submission queue to be addressed by kernel side polling from a dedicated kernel thread, removing the `io_uring_enter` call from the call stack completely and enabling I/O heavy programs to operate without the need for mode switch whatsoever, as long as the requests are frequent enough and the kernel doesn't get *bored*, a state describing the kernel having polled an empty submission queue for too long [4].

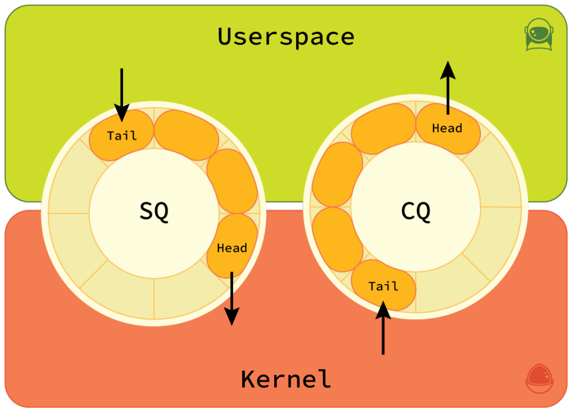


Figure 8: Illustration of io-urings underpinnings

io_uring reduces the overhead introduced by the system calls interface by lowering the number of mode switches associated with it under specific circumstances. Through its asynchronous nature it also creates less immediate interruptions to process execution, further lowering mode switch counts.

Implementing io_uring into existing code bases can be quite challenging. Drivers require extension to accommodate io_uring, so in order to adapt proprietary hardware, driver code must be rewritten. Also, creating asynchronous paradigms for requesting kernel controlled action comes with a multitude of benefits but comes with a major increase to complexity of implementation as asynchronous callback requires explicit handling to fetch return values upon completion, rather than the simple mechanism of assignment that a system call offers. This added complexity initially made Linus Torvalds, the creator and chief maintainer of the kernel suspicious (see: Figure 3.1), and should be kept in mind when evaluating io_uring for potential application.

”It will probably have subtle and nasty bugs, not just because nobody tests it, but because that’s how asynchronous code works - it’s hard.” - Linus Torvalds[3]

io_uring’s integration into the official kernel repository makes it available for all drivers implementing the corresponding fops entry. As the interface into io_uring communicates with the submission and completion infrastructure, attaching a driver back end is less complex than many other driver changes, requiring an addition to the fops but no fundamental change of driver behavior (see: Figure 9). The true complexity of using the interface is the adaptation to the asynchronous context.

```c
// ...

static struct file_operations fops = {
    // ...
    uring_cmd : logmodule_uring_cmd,
    // ...
};

static int logmodule_uring_cmd(struct io_uring_cmd *cmd, unsigned int issue_flags)
{
    /* command handling */

    // ...

    io_uring_cmd_done(cmd, 0, 0, issue_flags);
    return -EIOCBQUEUED;
}

// ...
```

Figure 9: Driver code adaptations for implementing io_uring tie-in

When compared to the simplicity of the `write()` system call regarded in the introduction (see: Figure 4), the implementation of this basic functionality requires a lot more knowledge of internal workings, making code difficult to read when not familiar with the operational underpinnings (see: Figure 10). Also, the kernel-side polling can become inefficient if the `sq_thread_idle` timeout, which controls the duration the kernel continues polling the submission queue after the last request, is set too large, as it may result in unnecessary CPU resource consumption when checking an empty queue[26].

```cpp
void UringBenchmark::BasicUringWrite()
{
    static const char buf[] = "buf";

    m_sqe = io_uring_get_sqe(&m_uring);

    if (!m_sqe)
    {
        std::cerr << "Failed to get SQE\n";
        return;
    }

    m_sqe->opcode = IORING_OP_WRITE;
    m_sqe->fd = m_fd;
    m_sqe->addr = (uintptr_t)buf;
    m_sqe->len = sizeof(buf);

    io_uring_submit(&m_uring);

    io_uring_wait_cqe(&m_uring, &m_cqe);

    std::cout << "uring return:" << m_cqe->res << "\n";

    io_uring_cqe_seen(&m_uring, m_cqe);
}
```

Figure 10: Hello World example of a simple wirte using io_uring

Compared to legacy interfaces like libaio or epoll, io_uring offers a unified and lower-latency interface for both file and socket I/O. Unlike domain-specific solutions such as SPDK or DPDK, it requires no specialized hardware or driver model, making it more suitable for general-purpose or embedded environments. Its reliance on memory mapping and optional kernel-side polling places it conceptually close to vFIO and other zero-copy user-kernel communication strategies.

io_uring offers an endpoint agnostic approach to performant and asynchronous I/O for use cases that demand frequent operations. Memory-mapped queue structures eliminate unnecessary data copying between user space and kernel space, while kernel-side submission queue polling, along with the ability to batch I/O requests from user space, minimizes the need for costly context switching into the kernel. In a trade off, io_uring inroduces an increase in implementation complexity as the asynchronous nature of its I/O paradigm means that engineers need to deal with the added intricacies of the asynchronous callback. io_uring has shown great potential when applied to both high load networking[25] and storage[7, 10, 43], but works best when applied in batch processing[57]. Properly abstracted io_uring can offer massive performance increases and presents a very interesting candidate for Televic's I/O stack.

3.2 eBPF

The eBPF originated from the *Berkley Packet Filter*, a network packet filtering scheme which aimed to run sandboxed just-in-time (JIT) compiled programs in a kernel embedded virtual machine to drastically improve filtering speeds[20]. From its narrowly focused beginnings, the *extended* version eBPF introduced an approach very closely mapped to emerging Instruction Set Architecture (ISA)[24], to utilize it's call-attached *modus operandi* to enable a more flexible and efficient execution model for a wider range of kernel operations. Formally, eBPF was added to the kernel in 2014 for version 3.15[50]. Since then it has grown to become an increasingly widely adopted technology, finding use cases in many more domains than just networking, including storage I/O[57], performance tracing[53], and even permission control within the kernel itself[21]. Modern eBPF serves as a convenient entry point into customizing kernel behavior from userspace during runtime, bypassing the slow trot of kernel feature innovations[20].

eBPF programs work by attaching themselves to kernel events such as system calls invocations, function calls, network events, and others (see: Figure 11)[20]. That means that eBPF programs are event-driven and only execute when the hook it is attached to is called. Since eBPF allows for user space injected code to run within the kernel, the proposed instructions have to be verified so as to not endanger kernel integrity. This verification is performed by the eBPF-verifier, which checks that the code is safe to execute within the kernel, ensuring that the code won't contain infinite loops or unchecked pointer dereferencing and won't access memory beyond its permitted range[45, p.15].

```
syscalls:sys_enter_accept                          [Tracepoint event]
syscalls:sys_enter_accept4                         [Tracepoint event]
syscalls:sys_enter_acct                            [Tracepoint event]
syscalls:sys_enter_add_key                         [Tracepoint event]
syscalls:sys_enter_adjtimex                        [Tracepoint event]
syscalls:sys_enter_arm64_personality               [Tracepoint event]
syscalls:sys_enter_bind                            [Tracepoint event]
syscalls:sys_enter_bpf                             [Tracepoint event]
```

Figure 11: Excerpt of a List of system calltracepoints an eBPF program could be attached to

eBPF also implements maps, key-value data storage objects that are accessible from user space and are the primary method of passing data to and from eBPF programs[44, p.20][20].

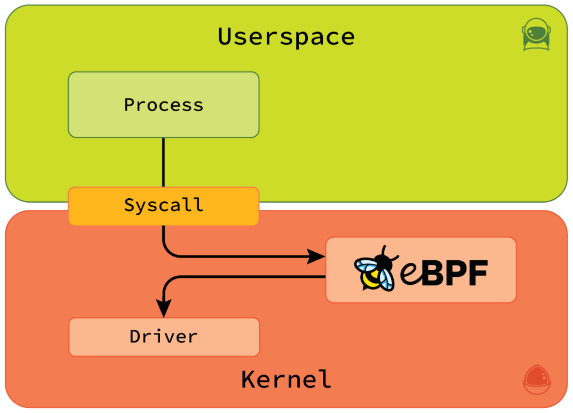


Figure 12: Illustration of a system call entry attached eBPF call

Since eBPF programs run within the kernel, they have direct access to kernel level functionality. This means in turn that the hooked functions can call kernel level functionality without incurring the costs for mode switchs[57]. The true strength of these programs therefor lie in *additional* action to kernel level functionality. Even though eBPF denies its acronym, claiming the term is now a moniker[20], its method betrays its roots. Packet filtering at the kernel level, as implemented by eBPF, is highly efficient because it allows decisions, such as dropping packets, to be made immediately after the callback from the Network Interface Card (NIC), minimizing unnecessary overhead. This approach is well-suited to other tasks that can benefit from similar early-stage, in-kernel decision-making, where performance is critical and low-latency actions are required.

The ease of implementation and ability to be dynamically loaded during runtime makes eBPF-Programs very versatile. Since the kernel is a complicated piece of software, it might often be easier to implement functionality using eBPF instead of attempting to patch the kernel. Because of eBPFs method of functioning, the interface demands a very specific use case meaning that adapting existing functionality to eBPF might be complex or completely infeasible. In addition to that, eBPF demands a specific compilation target, as the verifier can only work with certain byte code formats, resulting in only two programming languages currently being supported: C and Rust[45, p.19].

Overall eBPF offers a path into implementing high performance extensions of kernel level functionality. Although the context that eBPF programs exist in does not lend itself well to a generalized approach, the technique allows for a whole host of interesting possibilities. However, when evaluated against our original intention of optimizing agnostic driver access, eBPF shows little promise so far. Research efforts have attempted to extend the framework to facilitate a more direct interaction with system components[57], but until these efforts are brought to fruition and merged into the kernel, eBPF does not fit our use case.

3.3 DPDK & SPDK

The DPDK is a set of libraries and drivers designed to accelerate packet-processing tasks. First released by Intel in 2010, DPDK was intended to offer a software based alternative to Application-Specific Integrated Circuit (ASIC) chips that were designed to facilitate high performance networking. When CPUs became fast enough to compete with dedicated hardware solutions, the linux I/O stack bottleneck became the defining constraint for a migration away from ASIC reliant architectures. DPDK's libraries introduced a kernel bypass approach to NIC interaction combined with Poll Mode Drivers (PMDs), to overcome the performance limitations imposed by traditional Linux I/O[2, p.4]. The project has since transitioned to the Linux Foundation and is open source[18, 16].

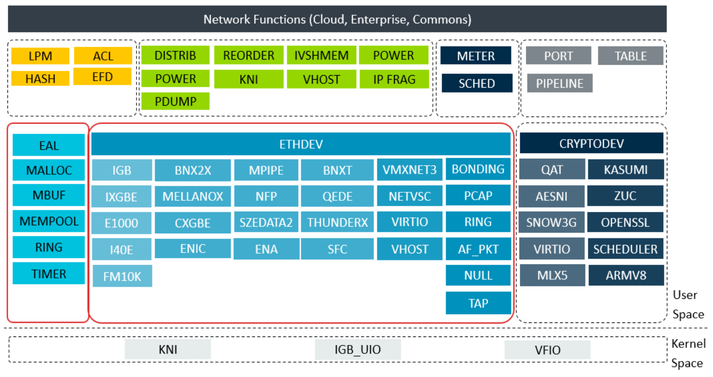


Figure 13: Listing of DPDK libraries and their contexts[32]

SPDK is the spiritual successor to the DPDK. Adopting the proofed methodologies tested in the networking landscape DPDK operates in, SPDK applies the same approaches to the world of memory I/O. With the trend for storage devices moving away from Hard Disk Drives (HDDs) and into faster solid state storage solutions such as NVMe SSDs, the proportion of time taken up by the operating system in relation to the time it took to write to storage became ever larger[54]. Where DPDK was developed to interact with NICs, SPDK was developed to increase speeds in storage interaction. Similar to DPDK, the software package comprising SPDK is being developed at Intel, but plans for its transition to the Linux Foundation are are already announced[51].

Both libraries are highly complex and share many of the same foundational principles (see Figure 13). While each is tailored to specific use cases, they are built upon similar core architectural concepts, especially in terms of how they handle high-performance I/O operations. DPDK is based on five core operational concepts that form the foundation of its functionality, enabling it to achieve efficient packet processing and network I/O (see Figure 14)[19, 16].

`EAL`	(Environment Abstraction Layer) hardware abstraction library enables hardware agnostic approach on a high level. The library facilitates setting up DPDK by handling Peripheral Component Interconnect (PCI) bus access, CPU feature identification, interrupt handling, memory management, etc.
`MBUF`	Sets up and manages large buffer pools, large amounts of memory that are prepared to take in packets from the network.
`MEMPOOL`	Huge page aggregation.
`RING`	Lockless ring buffer based inter-thread communication.
`TIMER`	Event trigger timing libraries.

Figure 14: DPDK core libraries

In addition to these paradigms DPDK relies PMDs, user space drivers that can be polled for status instead of relying on interrupt based notification[19]. SPDK in turn adopts many of the prooven methods applied by DPDK, reusing the `EAL` and `RING` components and copying DPDKs affinity for user space PMDs[54].

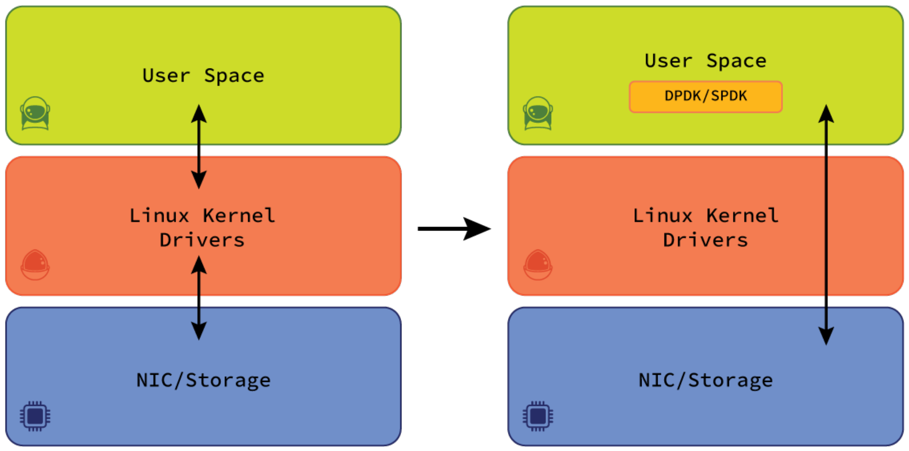


Figure 15: Illustration of dpdk and spdk working methodologies in contrast to standard Linux I/O

Both libraries achieve incredible performance gains and scalability in their respective domains[43, 6, 38]. Through a highly specialized stack focused on user space PMDs they achieve close to full kernel bypass, lowering the amount of incurred mode switchs to minimum and allowing for the advances in hardware in both domains to be unobstructed by the I/O system call interface and the overhead it introduces.

Since both SPDK and DPDK are libraries, integration into existing software is simple. Both technologies offer a simple API to achieve high performance, kernel bypass I/O, enabling developers to remain unburdened by the complexities of the implementation. Extending these libraries to fit specific use cases involves careful integration with the existing framework, ensuring that any changes or additions are compatible with the core functionality. This process can be complex, as developers need to account for the inner workings of the library and an extensive knowledge of other technologies used, as both libraries utilize frameworks like vFIO or Memory-Mapped IO (MMIO) [17].

DPDK and SPDK offer high performance I/O pipelines for networking and storage. The use of libraries makes implementation simple, granting access to a powerfull stack of functionalities as long as the libraries support the deployed hardware with a dedicated driver. Both libraries abstract from highly sophisticated implementations with ongoing development. The technology is highly interesting for applications with the need to access very high performant I/O quickly. However the narrow focus and the lack of easy extensibilty of both SPDK and DPDK makes them difficult to adapt to use cases that fall outside their originally intended design scope. When evaluating their use against the MVB card example that motivates our search for performant I/O, its clear that the lack of simple adaptability does not make the two libraries particularly feasible. In the context of embedded systems it should also be said that enabling 1Gb huge pages on systems with limited ram might require a lot of further deliberation and testing. Moreover, both technologies are primarily designed for high-throughput data center environments, making them disproportionately complex and resource-intensive for comparatively simple embedded system use cases such as our MVB card scenario.

3.4 vFIO

vFIO was an acronym but has, over the years, attempted to drop its misleading naming, similiar to eBPF, by overriding the naming scheme. The maintainer, Alex Williamson, proposed an alternative acronym overload in a talk in 2016, landing on *versatile framework for userspace I/O*[56]. It didn't stick. Merged into the kernel for release 3.6, vFIO was an improved, driver agnostic adaptation of the Kernel-based Virtual Machine (KVM) PCI pass-through mechanism that was used to grant virtual machines direct access to PCI-devices[15]. vFIO has since outgrown its original purpose of providing direct access to physical devices to user-mode hypervisors, and has become a secure framework for user space drivers. It has become the de-facto standard for device pass-through, combining the performance of user space device access with the security of kernel mediation[34, 30, 22].

vFIO maps drivers to user space. Analogous to previous techniques this bypasses the need to interact with kernel I/O stack and allows for user land applications to interact with the hardware without the need for mode switch. In addition vFIO utilizes a custom Direct Memory Access (DMA) pipeline. A critical improvement to earlier device-passthrough methods is vFIOs integration of a user space MMU, the `IOMMU`, wich maps DMA requests, ensuring that devices can only access the correct memory. This allows vFIO to offer the advantages of direct device access without having to trade away process memory isolation.

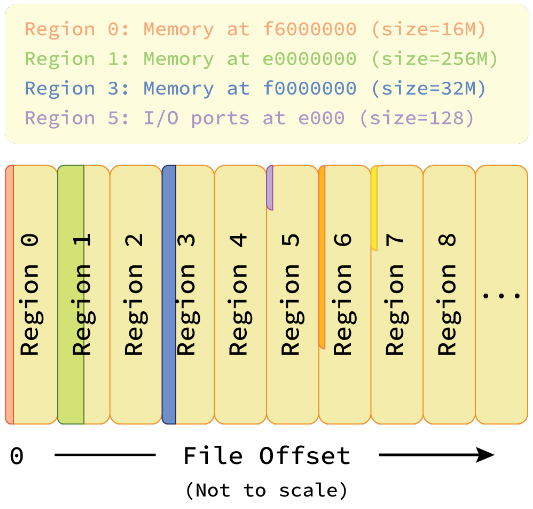

Region 0: Memory at f6000000 (size=16M)
Region 1: Memory at e0000000 (size=256M)
Region 3: Memory at f0000000 (size=32M)
Region 5: I/O ports at e000 (size=128)
Region 0
Region 1
Region 2
Region 3
Region 4
Region 5
Region 6
Region 7
Region 8
...
0
File Offset
(Not to scale)

Figure 16: Illustration of Memory mapping MMIO acces to a driver into process memory

Integrating vFIO into an existing software product is non-trivial and involves both hardware prerequisites and several low-level configuration steps. Firstly, the system must support an Input-Output Memory Management Unit (IOMMU), a MMU addressable from user space, with interrupt remapping capabilities. This typically requires setting platform-specific kernel parameters at boot (e.g., `intel_iommu=on` for Intel systems or `amd_iommu=on` for AMD) and enabling corresponding options in the Basic Input/Output System (BIOS)/Unified Extensible Firmware Interface (UEFI). Devices addressed from vFIO will then be unbound from their respective drivers and rebound to vFIOs `vfio-pci` module, which detaches the device from the kernel and exposes it as a file descriptor. From here, programs that want to interact with the device have to bind the file descriptor, retrieve its MMIO mapping and remap the devices interrupts[56, 34].

Although this integration unlocks near bare-metal performance and fine-grained control, it shifts significant responsibility to the developer. Common kernel subsystems such as power management or shared access mediation must now be reimplemented and manually managed. Moreover, vFIO enforces exclusivity. This means that only one process or virtual machine may own the device at any given time. Furthermore, reliance on the existence of an `IOMMU` and interrupt remapping makes vFIO hardware dependent, as only specific architectures support such features.

vFIO is a foundational corner stone of software products such as the Quick EMUlator (QEMU) and KVM, the data and storage plane optimization libraries DPDK and SPDK, and GPU-passthrough technologies such as NVIDIA `vGPU` or Looking Glass[30]. Direct use of vFIO is highly complex and demands a deep understanding of kernel level abstractions, making the use of vFIO by proxy far easier than direct interaction. vFIO offers memory managed DMA but lacks other kernel security features such as multi user mediation. This implies that an integration of the MVB-card driver would necessitate a complex adaption of the driver to sustain user space execution compatibility. These demands are likely to make it a badly suited candidate for integration with Televics existing software products and the MVB card.

MMIO and mmap

In Linux, memory-mapped I/O (MMIO) allows devices to map their registers and memory directly into the address space of user-space processes, enabling efficient and low-latency access to hardware. This is typically done using the mmap system call, which allows user-space applications to directly interact with device memory without needing to rely on kernel-level intervention. While vFIO provides a secure, controlled environment for using MMIO by managing device access through user space and leveraging the `IOMMU` for address translation and isolation, MMIO can also be used independently via `mmap`. This approach provides a simpler, though less secure, method of memory access and is commonly used in scenarios where direct device interaction is needed without the complexity of vFIO. However, in both cases, the responsibility for ensuring proper memory access and managing device interactions lies with the user-space application, and developers must account for the associated complexities.

3.5 Common Optimization Methods

When evaluating various optimization techniques surrounding the kernel I/O stack, two methods are particularly prevalent in the examples we have covered so far: direct, low mediation, device passthrough via memory remapping and ringbuffer data structures that utilize polling to enable lockless interaction between user space processes and drivers. These approaches minimize overhead and maximize performance by reducing the need for costly context switches and locking mechanisms. As we transition into the benchmarking section, we will focus on evaluating two prominent I/O optimization methods, DMA as implemented by MMIO via `mmap` and io_uring, both of which leverage these techniques to achieve significant performance improvements. By comparing these methods in terms of their efficiency, scalability, and impact on system resources, we will assess their suitability for high-performance I/O operations in modern computing environments and attempt to ascertain whether these techniques can be effectively applied to optimize the interaction with the MVB bus in 's embedded devices, enhancing data throughput and minimizing latency.

4 Benchmarking and Results

The evaluation below aims to quantify the performance benefits of different kernel bypass mechanisms in embedded systems by measuring the latency, throughput and cpu utilization under synthetic I/O workloads, aimed to mimic high frequency interactions against traditionally kernel controlled devices. It compares the write throughput of `systemcall`, `io_uring` and `mmio` via `mmap`. Benchmarking the performance of the techniques against one another will grant insight into how much performance overhead system calls introduce on a standard embedded system and how the polling and memory mapping techniques implemented by MMIO and io_uring perform under load.

4.1 Experimental Setup

To evaluate the performance of different optimization techniques, we will quantify the throughput of a series of `writes` to a unified endpoint. First, the techniques will be tested against a stub driver. Next, the same process will be repeated with file writes to a USB stick mounted into the Raspberry Pi filesystem. In both tests, one million write operations are performed, measuring the time spent in user space and kernel space, the overall throughput, and the average time between write operations. In addition to benchmarking against two different endpoints, the tests are conducted using varying data sizes, with one set of tests employing 4-byte-sized writes and another utilizing 4KB-sized writes.

As a platform we chose the Raspberry Pi 4, a popular consumer grade Single-Board Computer (SBC), which contains an Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit processor clocked at 1.8 GHz and 4GB of Random Access Memory (RAM), running the platform specific Debian bookworm clone Raspberry Pi OS, underpinned by an adapted kernel, branched from version 6.12.25[42, 41, 40]. This device is less powerful than Televic's embedded systems, and as a result, any overhead introduced by the optimization techniques is likely to be more pronounced. With limited processing power and resources, the impact of hardware-related bottlenecks will be amplified, making it an effective environment for evaluating how these techniques can optimize performance in more resource-constrained systems.

The multi-purpose stub-driver acts as a `write`-sink. This driver offers memory mapping capabilities, is compatible with io_uring and opens a pseudo device that can be addressed as though addressing a serial port or PTY. It also implements tracepoints at it functions (see; Figure 17), allowing insight into exactly when the functionality is being called by the benchmark, and how many times the included tracepoints are hit. This driver was compiled into the Raspberry Pi specific kernel .

```c
static ssize_t logmodule_write(struct file *, const char __user *, size_t len,
    loff_t *)
{
    trace_write(current->pid);

    LM_LOG("wrt \t[ktime: %lld][write_n: %d][pid: %d]\n", ktime_get(), write_n,
    current->pid);
    write_n += 1;
    return len;
}
```

```c
TRACE_EVENT(write,
            TP_PROTO(pid_t pid),
            TP_ARGS(pid),
            TP_STRUCT__entry(
                __field(pid_t, pid)),
            TP_fast_assign(
                __entry->pid = pid),
            TP_printk("write: %d", __entry->pid));
```

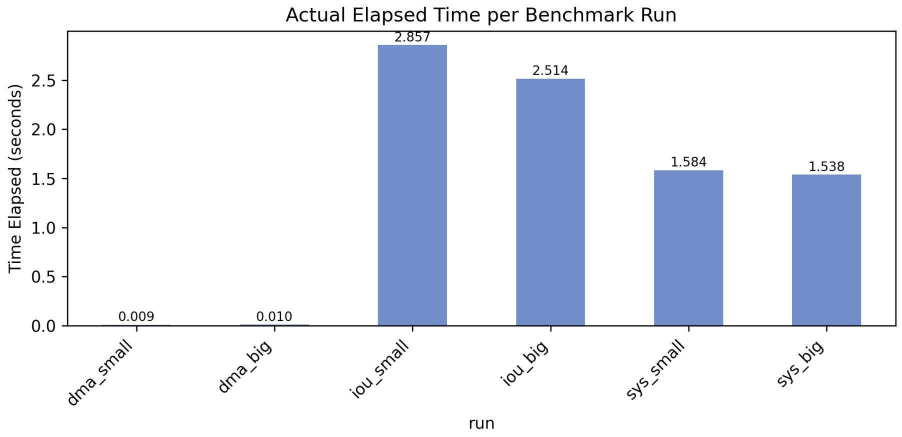
Figure 17: Logmodule Driver `write` and corresponding tracer

In order to provide a fair comparison, the benchmark for io_uring is conducted using a multi-threaded approach, with separate producer and consumer threads. If io_uring were evaluated using the same methodology as system call, where each task is waited on to complete before executing the next, the asynchronous nature of io_uring would not be fully utilized (see: Figure 36).

To track the performance of our benchmark we utilize the Linux performance measuring tool `perf`, a dynamic tracing tool that will allow us insight into CPU performance counters, tracepoints and kernel probes. It is a system profiling tool that can be attached to the tracepoints integrated in the functions of the stub driver endpoint, enabling us to measure exact interaction timepoints. Additionally, enabling system call tracepoints while compiling the kernel, allows to attach to system call events, thereby providing detailed visibility into the mode switch process. In this configuration, `perf`, in conjunction with the implemented tracepoints, offers high-resolution, timestamped insights while maintaining minimal observation overhead.

4.2 Results

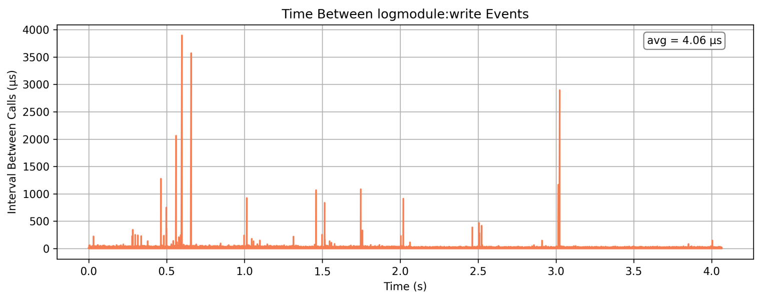
In this section, we present the results of the benchmarking experiments aimed at evaluating the performance of different I/O optimization techniques, specifically focusing on DMA via `mmap`/MMIO, io_uring, and traditional system calls. The results reveal that DMA significantly outperforms the other methods in both throughput and latency, showcasing its potential for high-performance I/O operations. On the other hand, io_uring, despite its promise for efficient asynchronous I/O, performed surprisingly inefficiently in the context of these tests, particularly in comparison to more traditional methods. Interestingly, traditional system call delivered performance that was notably better than initially expected.



Run	User Time	Sys Time	Total Time	User %	Sys %
systemcall 4b	0.232	1.351	1.584	14.6%	85.3%
systemcall 4kb	0.255	1.278	1.538	16.6%	83.1%
io_uring 4b	0.664	7.238	2.857	23.3%	253.4%
io_uring 4kb	0.670	6.947	2.514	26.7%	276.3%
dma 4b	0.004	0.004	0.009	45.5%	45.5%
dma 4kb	0.004	0.004	0.010	41.1%	41.1%

Figure 18: Performance Benchmarking: One Million Writes to System Endpoint

In the benchmarking against the stub driver endpoint we can see that io_uring requires a lot more CPU resources than both MMIO and system calls, spending 6.9 seconds of CPU-time, writing to the endpoint. Since the technique spawns a dedicated kernel thread polling the submission queue, and was benchmarked against a multi-threaded approach, this is to be expected but also means that io_uring is far less efficient with system resources than the traditional system call (see: Figure 18 and Figure 21). io_uring also performs worse on average write latency, which is highly surprising, given the dedicated thread who's sole purpose it is to write to the driver (see: Figure 19), also showing the biggest increase in latency in correlation with the size of the payload.



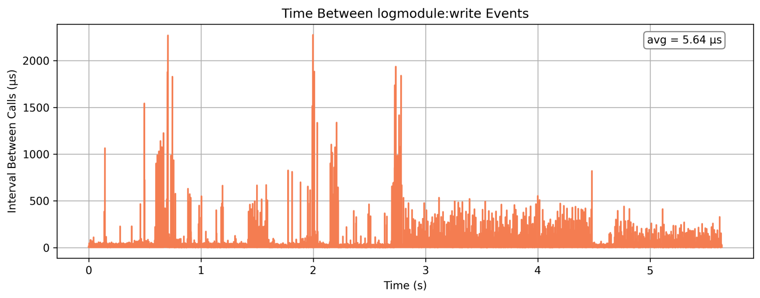
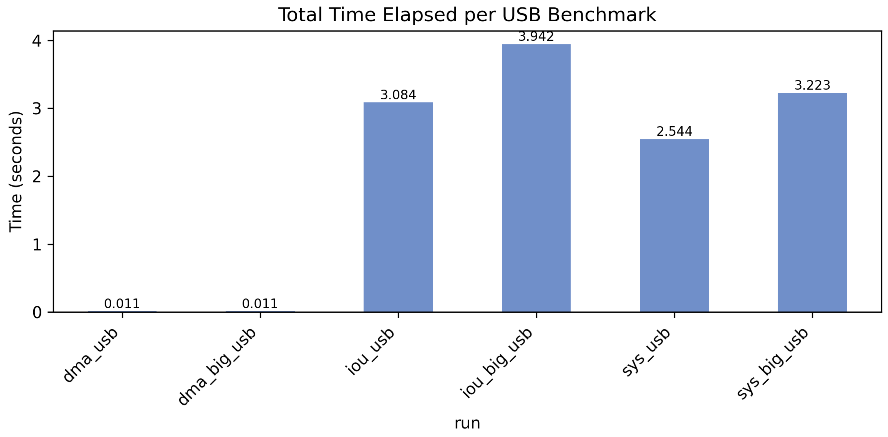


Figure 19: io_uring: Time Between Writes for 1M/4B (Upper) and 1M/4KB (Lower)

We should also note that in both runs against the stub driver and the hardware endpoint, our measuring tool had difficulties keeping up with the MMIO approach due to the lack of available tracepoints, evaluating the `Sys/User` percentages drastically differently ever time. The results displayed here, suggesting an almost equal split amongst the domains, was the most common outcome but is not to be taken as a consistent result (Figure 21 & Figure 20 & Figure 18).



Run	User Time	Sys Time	Total Time	User %	Sys %
systemcall 4b to USB	0.452	2.092	2.544	17.8%	82.2%
systemcall 4kb to USB	0.467	2.750	3.223	14.5%	85.3%
io_uring 4b to USB	0.764	7.850	3.084	24.8%	254.6%
io_uring 4kb to USB	0.676	10.620	3.942	17.2%	269.4%
dma 4b to USB	0.004	0.004	0.011	39.7%	39.7%
dma 4kb to USB	0.004	0.004	0.011	40.4%	40.4%

Figure 20: Performance Benchmarking: One Million Writes to Hardware Endpoint

Interestingly writing to the USB Device, initially chosen because it presents a slower device than the system memory the stub driver resides in, does not introduce any overhead to the MMIO approach at all, punishing the system call benchmark proportionally the hardest (see: Figure 20 and Figure 18), perhaps giving an inkling to io_urings performance potential for larger writes.

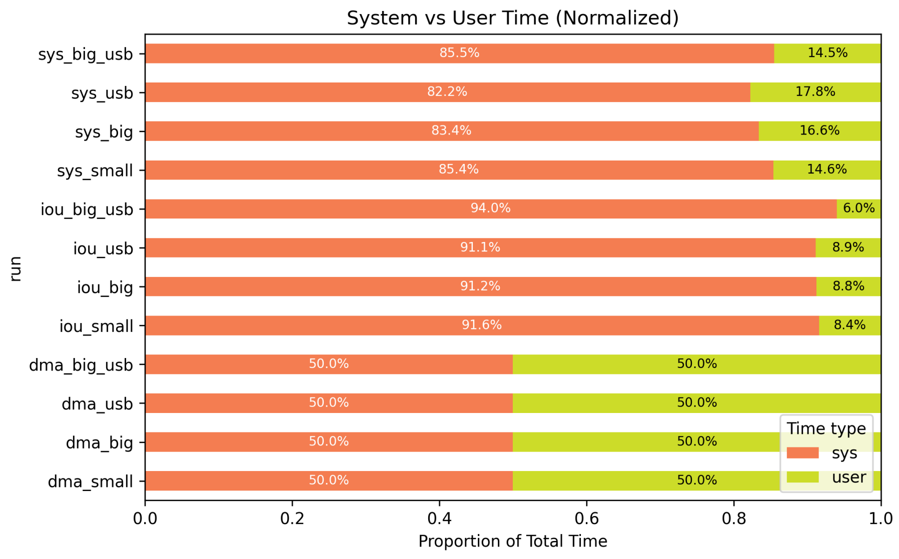


Figure 21: Normalized System/User Space Proportions for Write Operations

4.3 Analysis and Discussion

Overall the data seems to suggest that the system call mechanism is plenty performant, managing to outperform the safe and asynchronous io_uring consistently across our benchmarking. Although personally slightly disappointing, it is perhaps indicative of the the fact that the overhead introduced by system calls's incurred mode switch is far less detrimental than the overhead introduced by io_uring's additional infrastructure, requiring thread spawning and buffer mapping. Perhaps, in addition, our benchmarking approach—focused on consecutive writes—did not offer an environment in which io_uring could fully demonstrate its effectiveness. We can gleam from the overall runtime duration staying roughly the same for the 4b and 4kb writes, that io_uring's memory mapped queue buffers enable efficient data transfer into the kernel and back, bypassing the user space copy that system calls rely on.

The true takeaway from this effort is the incredible performance increase of MMIO, outperforming io_uring and system calls by factors of up to 100. It should be mentioned at this point that writing come characters into a buffer is a use case DMA can very easily handle efficiently, our benchmarking scheme not offering a sufficiently complex interaction space to properly assert how difficult full user space driver integration could become when handling more complicated needs. Also, as previously mentioned, the MMIO based benchmarked elegantly sidestepped the kernel tracers that were laid to collect data from both other methods, bypassing the overhead that these measuring techniques introduce, further giving it a slight advantage.

In summary, while the benchmarking results show that traditional system call mechanisms remain highly performant, especially when compared to io_uring, it is clear that each method has its own strengths and limitations depending on the use case. The performance of io_uring appears to be hindered by the overhead of additional infrastructure and thread management, which may not be well-suited for our benchmarking scenario focused on consecutive writes. However, its memory-mapped queue buffers do show promise for efficient data transfer, especially in scenarios where I/O operations are more varied. Meanwhile, the remarkable performance of MMIO in this context highlights the significant potential of direct device access, offering exceptional throughput with minimal overhead.

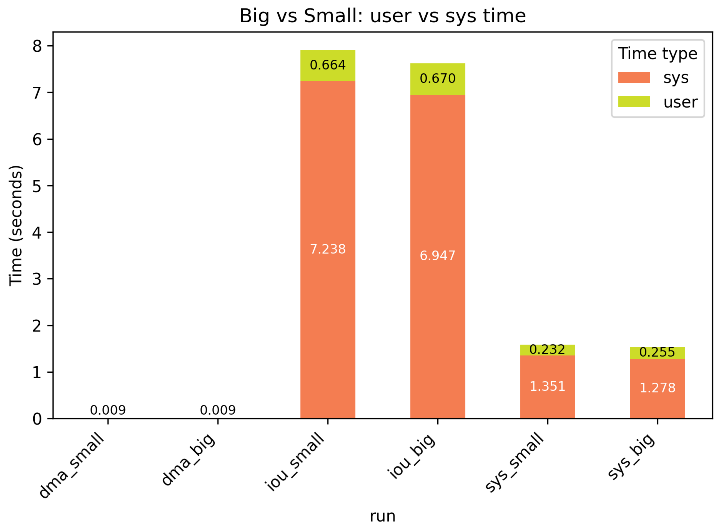


Figure 22: Performance Benchmarking: One Million Writes to System Endpoint (total CPU Time)

5 Conclusion

Throughout this thesis, we have taken a look at the underpinnings of the Linux I/O stack and evaluated efforts to optimize it for their complexity, their ability to be introduced to existing software, their generalizability in approach and their applicability to agnostic device access on embedded systems. We have chose representative methodologies, covering by their representation, the most prominently applied methods in the field and have measured their performance against the Linux system call to evaluate their performance.

We have found that on low powered systems, the performance gains promised by the asynchronous I/O interface io_uring fall short when evaluated against the system call default within the narrow constrained test case of writing to both driver as well as peripheral storage and that, for high performance throughput, MMIO is not without reason the technology underpinning the most performance focused libraries and stacks. Also we have found that the plethora of techniques by which the Linux I/O stack is optimized in high performance systems is not easily transfered to the embedded context without consideration. While no single method emerged as a universal solution, our findings clearly illustrate that performance-critical applications, particularly in embedded contexts, may benefit significantly from bypassing the kernel's I/O stack, provided the complexity and risk are carefully managed.

Our findings seem to indicate that if performance for the MVB card is desperately needed, the best option is to brave the complicated world of user space drivers and do away with the system call interface by mapping existing drivers closer to the software solutions that require their services. But mostly, our findings indicate that a lot more research is necessary to make an informed decision.

The benchmarking performed in the context of this thesis truly creates more questions than it provides answers. Why did io_uring under perform against our initial expectations despite completely bypassing the incurrence of mode switches? How much did our chosen DMA method benefit from the lack of tracepoints in its path? How would older optimization techniques such as or epoll perform in contrast to the newer io_uring. The ideation on unanswered questions seems to drift into the endless void.

In future work, a number of promising directions are already teased by our despair in the former paragraph. Firstly a more robust and comprehensive benchmarking scheme should be developed or found, including far more techniques to contrast against. It is to be assumed that an easier method could have been chosen to build the benchmarking scheme presented in this thesis, enabling a far more interesting and in depth analysis of the kernel / user space dichotomy. Furthermore the authors affinity for the programming language Rust and the convergence of timings, with the language inching ever closer to to being officially included in the Linux kernel, promises to offer a rich field of research, pairing the inherent memory safety offered by Rust with the performance of MMIO or the easy extensibility of the kernel through eBPF. Also, having mentioned eBPF, the development of a much more robust tracing ecosystem, including user space tracepoints holds the promise to develop true insight into how techniques utilizing full kernel bypass, such as the DMA technique using `mmap` presented in this thesis, operate under the hood.

In closing, this thesis began with a simple ambition: to meaningfully improve I/O performance on resource constrained systems. Along the way, it became clear fast, that the answer to this question is far from straight forward. Like many works in systems research, this thesis raises as more questions than it answers. Hopefully this is where its value lies: in exposing the nuanced landscape of Linux I/O optimization for embedded devices, and in providing a foundation for future efforts seeking to design more efficient, predictable, and adaptable I/O subsystems.

I sincerely thank Televic and its dedicated team for their time, support, and for providing the platform that made this work possible.

The project files aswell as the benchmarking source code can be found in the FU Gitlab repository.

References

[1] Vaggelis Atlidakis et al. “POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing”. In: *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys ’16)*. London, United Kingdom: ACM, 2016. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901350. URL: https://dl.acm.org/doi/pdf/10.1145/2901318.2901350.

[2] AvidThink and The Linux Foundation. *Myth-busting DPDK in 2020*. Whitepaper. Rev B. The Linux Foundation, 2020. URL: https://nextgeninfra.io/wp-content/uploads/2020/07/AvidThink-Linux-Foundation-Myth-busting-DPDK-in-2020-Research-Brief-REV-B.pdf.

[3] Jens Axboe. *Faster IO through io_uring*. https://archives.kernel-recipes.org/wp-content/uploads/2025/01/kernel-recipes-2019-axboe.pdf. Presentation at Kernel Recipes 2019, Sep 26th. Software Engineer, Facebook. 2019.

[4] Jens Axboe. *io_uring Whitepaper*. Tech. rep. Version 0.4, 2019-10-15. kernel.dk, 2019. URL: https://kernel.dk/io_uring.pdf.

[5] Eli Bendersky. *Measuring Context Switching and Memory Overheads for Linux Threads*. https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/. Accessed 2025-05-01. 2018.

[6] Hao Bi and Zhao-Hun Wang. “DPDK-based Improvement of Packet Forwarding”. In: *Proceedings of the ITA 2016 Conference*. Vol. 7. TM Web of Conferences. Open access under Creative Commons Attribution License 4.0. EDP Sciences, 2016, p. 01009. DOI: 10.1051/itmconf/20160701009.

[7] Matias Bjørling et al. “Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems”. In: *Proceedings of the 6th ACM International Systems and Storage Conference (SYSTOR ’13)*. Haifa, Israel: ACM, 2013. ISBN: 978-1-4503-2116-7. DOI: 10.1145/2485732.2485734. URL: https://kernel.dk/blk-mq.pdf.

[8] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. 3rd. Sebastopol, CA, USA: O’Reilly Media, Inc., 2005. ISBN: 978-0-596-00565-8.

[9] CERN. *Inauguration of the IBM 709*. https://timeline.web.cern.ch/inauguration-ibm-709. Accessed 2025-05-01.

[10] Andrea Cerone et al. “Understanding Modern Storage APIs: A Systematic Study of libaio, SPDK, and io_uring”. In: *SYSTOR*. 2022. URL: https://atlarge-research.com/pdfs/2022-systor-apis.pdf.

[11] David L. Chandler. *Behind the Scenes of the Apollo Mission at MIT*. https://news.mit.edu/2019/behind-scenes-apollo-mission-0718. Published July 18, 2019, MIT News Office. 2019.

[12] The kernel development community. *System Calls*. © Copyright The kernel development community. URL: https://linux-kernel-labs.github.io/refs/heads/master/lectures/syscalls.html.

[13] *Concepts Overview: Linux Memory Management Documentation*. Linux Kernel Documentation, version 6.4. Accessed 2025-05-01. 2023.

[14] Jonathan Corbet. "Ringing in a new asynchronous I/O API". In: *LWN.net* (Jan. 15, 2019). URL: https://lwn.net/Articles/776703/.

[15] Jonathan Corbet. "Safe Device Assignment with VFIO". In: *LWN.net* (Jan. 3, 2012). Accessed 2025-05-01. URL: https://lwn.net/Articles/474088/.

[16] *DPDK Documentation, Version 25.03.0.* Accessed 2025-05-01. DPDK Project. 2025.

[17] *DPDK Linux Drivers Guide.* Accessed 2025-05-01. Part of the DPDK Getting Started Guide for Linux. 2025.

[18] DPDK Project. *DPDK: Data Plane Development Kit.* https://github.com/DPDK/dpdk. Accessed 2025-05-01. Licensed under BSD-3-Clause (core) and GPL-2.0 (kernel components).

[19] Cristian Dumitrescu. *2.6 DPDK Overview.* https://www.youtube.com/watch?v=0G6u409cSos&list=PLOoYNH2m9j3cYDhY4ScQ6qaleMcMfuAJa. Presented at FD.io /dev/boot, June 2016, Paris, France. Intel Software Architect for Packet Processing. 2016.

[20] eBPF.io authors. *What is eBPF?* The content of the ebpf.io website is licensed under a Creative Commons Attribution 4.0 International License. 2024. URL: https://ebpf.io/what-is-ebpf/.

[21] Jake Edge. "A seccomp overview". In: *LWN.net* (Sept. 2, 2015). Presented at the Linux Plumbers Conference. Licensed under CC BY-SA 4.0. URL: https://lwn.net/Articles/656307/.

[22] Paul Emmerich et al. *User Space Network Drivers.* https://arxiv.org/pdf/1901.10664. 2019.

[23] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95).* M.I.T. Laboratory for Computer Science. Cambridge, MA, USA: ACM, 1995. DOI: 10.1145/224056.224076.

[24] Matt Fleming. "A Thorough Introduction to eBPF". In: *LWN.net* (Dec. 2, 2017). Accessed: 21.11.2024. URL: https://lwn.net/Articles/740157/.

[25] Donald Hunter. *Why You Should Use io_uring for Network I/O.* https://developers.redhat.com/articles/2023/04/12/why-you-should-use-iouring-network-io. Published April 12, 2023. Last updated August 14, 2023. 2023.

[26] Shuveb Hussain. *Submission Queue Polling.* https://unixism.net/loti/tutorial/sq_poll.html. Accessed 2025-05-01. 2020.

[27] Shuveb Hussain. *What is io_uring?* © Hussain, Shuveb 2020, Accessed: 10.11.2024. URL: https://unixism.net/loti/what_is_io_uring.html.

[28] IBM. *Time-sharing.* https://www.ibm.com/history/time-sharing. Accessed 2025-05-01.

[29] Odun-Ayo Isaac et al. "An Overview of Microkernel Based Operating Systems". In: *IOP Conference Series: Materials Science and Engineering.* Vol. 1107. Presented at the International Conference on Engineering for Sustainable World (ICESW 2020), August 10–14, 2020, Ota, Nigeria. IOP Publishing Ltd, 2021, p. 012052. DOI: 10.1088/1757-899X/1107/1/012052.

[30] Neo Jia and Kirti Wankhede. *vGPU on KVM – A VFIO Based Framework.* https://www.linux-kvm.org/images/5/59/02x03-Neo_Jia_and_Kirti_Wankhede-vGPU_on_KVM-A_VFIO_based_Framework.pdf. 2016.

[31] Michael Kerrisk. *syscall(2) — Linux manual page*. `https://man7.org/linux/man-pages/man2/syscall.2.html`. HTML rendering created 2025-02-02. 2025. (Visited on 05/01/2025).

[32] Joyce Kong. *DPDK Optimization on Arm*. `https://community.arm.com/arm-community-blogs/b/tools-software-ides-blog/posts/dpdk-optimization-on-arm`. Published May 3, 2022. Accessed 2025-05-01. 2022.

[33] Chuanpeng Li, Chen Ding, and Kai Shen. "Quantifying the Cost of Context Switch". In: *ExpCS*. Permission required for redistribution. ACM 978-1-59593-751-3/07/06. San Diego, CA: ACM, 2007. DOI: `10.1145/XXXXX.XXXXX`. URL: `https://www.usenix.org/legacy/events/expcs07/papers/2-li.pdf`.

[34] Linux Kernel Documentation. *VFIO – Virtual Function I/O*. `https://www.kernel.org/doc/html/v5.4/driver-api/vfio.html`. 2020.

[35] *Linux Kernel Teaching: Interrupts*. Accessed 2025-05-01. Part of Linux Kernel Teaching materials. 2025.

[36] Till Miemietz et al. "Fast privileged function calls". In: *11th Workshop on Systems for Post-Moore Architectures (SPMA)(Rennes, France)*. 2022. URL: `https://fis.tu-dresden.de/portal/files/17730329/202204_Miemietz_SPMA_FastCalls.pdf`.

[37] OpenAI. *ChatGPT-4o Language Model*. Used as an assistant for wording, ideation, BibTeX entry generation, glossary entry creation, and bug duck-style counterweight thinking during the research and writing process. 2025.

[38] JuGeon Pak and KeeHyun Park. "A High-Performance Implementation of an IoT System Using DPDK". In: *Applied Sciences* 8.4 (2018), p. 550. DOI: `10.3390/app8040550`. URL: `https://www.mdpi.com/2076-3417/8/4/550`.

[39] *Processor privilege levels, execution privilege, and access privilege*. `https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Application-Level-Memory-Model/Access-rights/Processor-privilege-levels--execution-privilege--and-access-privilege?lang=en`. Accessed 2025-05-01. 2025.

[40] Raspberry Pi Foundation. *Raspberry Pi Linux Kernel Source Tree (rpi-6.12.y)*. `https://github.com/raspberrypi/linux/tree/rpi-6.12.y`. Accessed 2025-05-01.

[41] Raspberry Pi Ltd. *Raspberry Pi 4 Product Brief*. `https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf`. Accessed 2025-05-01. 2023.

[42] Raspberry Pi Ltd. *Raspberry Pi OS Documentation*. Accessed 2025-05-01. Licensed under CC BY-SA 4.0. 2025.

[43] Zebin Ren and Animesh Trivedi. "Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io_uring". In: *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '23)*. Rome, Italy: ACM, 2023, pp. 1–11. ISBN: 979-8-4007-0081-1/23/05. DOI: `10.1145/3578353.3589545`. URL: `https://atlarge-research.com/pdfs/2023-cheops-iostack.pdf`.

[44] Liz Rice. *Learning eBPF*. 1st. Part of a collaboration between O'Reilly and Isovalent. Sebastopol, CA, USA: O'Reilly Media, Inc., 2023. ISBN: 978-1-098-13887-5.

[45] Liz Rice. *What Is eBPF?* Part of a collaboration between O'Reilly and Isovalent. Sebastopol, CA, USA, 2022.

[46] Peter Jay Salzman et al. *The Linux Kernel Module Programming Guide (updated for 5.0+ kernels)*. Topics: C, Linux, documentation, kernel, Linux kernel module, device driver. 2025. URL: https://sysprog21.github.io/lkmpg/.

[47] Georg Sauthoff. *On the Costs of Syscalls*. https://gms.tf/on-the-costs-of-syscalls.html. Accessed 2025-05-01, published Mon 30 August 2021. 2021.

[48] Satyajit Sinha. *State of IoT 2024: Number of Connected IoT Devices Growing 13% to 18.8 Billion Globally*. https://iot-analytics.com/number-connected-iot-devices/. Published September 3, 2024. IoT Analytics. 2024.

[49] *Standard Streams*. https://en.wikipedia.org/wiki/Standard_streams. Last edited on 12 February 2025. Licensed under CC BY-SA 4.0. 2025.

[50] Alexei Starovoitov. *net: filter: rework/optimize internal BPF interpreter's instruction set*. Linux kernel source tree commit bd4cf0ed331a275e9bf5a49e6d0fd55dffc551b8. Committed by David S. Miller on 2014-03-31. Available at: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bd4cf0ed331a275e9bf5a49e6d0fd55dffc551b8. 2014.

[51] *Storage Performance Development Kit Documentation*. Accessed 2025-05-01. SPDK Project. 2025.

[52] *System Calls. Linux Kernel Labs Documentation*. Explains Linux system call mechanism (user-kernel transition, register saving, etc.) 2021. URL: https://linux-kernel-labs.github.io/lectures/syscalls.html.

[53] Marcos A. M. Vieira et al. "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications". In: *ACM Comput. Surv.* 53.1 (Feb. 2020). ISSN: 0360-0300. DOI: 10.1145/3371038. URL: https://doi.org/10.1145/3371038.

[54] Benjamin Walker. *SPDK: Building Blocks for Scalable, High Performance Storage Applications*. https://www.snia.org/sites/default/files/SDC/2016/presentations/performance/BenjaminWalker_SPDK_Building_Blocks_SDC_2016.pdf. Presented at SNIA Storage Developer Conference (SDC), 2016. Intel Corporation. 2016.

[55] Stephen R. Walli. "The POSIX Family of Standards". In: *StandardView* 3.1 (1995). Includes references to IEEE Std 1003.1-1990 and NIST POSIX email server (1994), pp. 17–23. DOI: 10.1145/210308.210315. URL: https://dl.acm.org/doi/pdf/10.1145/210308.210315.

[56] Alex Williamson. *An Introduction to PCI Device Assignment with VFIO*. https://www.linux-kvm.org/images/5/54/01x04-Alex_Williamson-An_Introduction_to_PCI_Device_Assignment_with_VFIO.pdf. Presentation by Alex Williamson (Red Hat). Video available at https://www.youtube.com/watch?v=WFkdTFTOTpA&t=1435s. 2016.

[57] Yu Jian Wu et al. "BPF for Storage: An Exokernel-inspired Approach". In: *arXiv preprint* (2021). URL: https://arxiv.org/abs/2102.12922.

[58] Ziya Yang. *NVMe-oF TCP Transport in SPDK*. https://snia.org/educational-library/spdk-based-user-space-nvme-over-tcp-transport-solution-2019. Presentation at Storage Developer Conference, September 26, 2019. Covers SPDK's implementation of NVMe-oF TCP transport and performance improvements using user-space networking techniques. 2019.

Glossary

API Application Programming Interface. 5, 23

ASIC Application-Specific Integrated Circuit. 21

BIOS Basic Input/Output System. 24

context switch The process by which the operating system saves the state of a currently running process and loads the state of another.. 8

CPU Central Processing Unit. 10, 12, 21, 22, 27, 28

CPU pipeline A technique used in modern processors where multiple instruction processing stages (such as fetch, decode, execute) are overlapped to improve instruction throughput and overall performance. 10

critical path The longest sequence of dependent operations in a system or process, determining the minimum time required to complete a task.. 6

DAU Dumbest Assumable User. 6

DMA Direct Memory Access. 23–25, 27, 31, 33

DPDK Data Plane Development Kit. 15, 18, 21–24

driver A low-level software component that enables communication between the operating system and hardware devices.. 4, 8, 25

eBPF Extended Berkeley Packet Filter. 15, 19, 20, 23, 33

edge computing A distributed computing paradigm that brings computation and data storage closer to the location where it is needed, improving response times and saving bandwidth. 6

epoll A scalable I/O event notification mechanism in the Linux kernel that allows efficient monitoring of multiple file descriptors for readiness to perform I/O. 13, 18, 33

exokernel An operating system kernel that exposes hardware resources directly to applications, allowing them to manage those resources themselves, offering maximum flexibility and performance. 4

ext4 The fourth extended file system used in Linux, known for its improved performance, support for large volumes and files, journaling capabilities, and backward compatibility with ext3. 11

fops file operations struct. 11, 16

glibc GNU C Library. 9

HDD Hard Disk Drive. 21

HPC High-Performance Computing. 6

huge page A memory management feature that uses larger-than-standard memory pages (e.g., 2MB or more) to reduce TLB misses and improve performance in memory-intensive applications. 10

huge page A memory management feature that uses larger-than-standard memory pages (typically 2MB or more) to reduce TLB misses and improve performance for memory-intensive applications. 22

interrupt A signal sent by hardware or software to the processor to indicate an event that needs immediate attention, temporarily halting the current execution flow to handle the event. 12

io_uring Ringbuffer based asynchronous I/O interface for Linux. 13, 15–18, 25–28, 30–33

IOMMU Input-Output Memory Management Unit. 24

IoT Internet of Things. 6

ISA Instruction Set Architecture. 19

JIT just-in-time. 19

kernel space The area of memory where the operating system's kernel runs, with full access to hardware and system resources. 4

KVM Kernel-based Virtual Machine. 23, 24

libaio Linux-native asynchronous I/O library. 15, 18

logical address An address generated by the CPU during program execution, typically referring to a location in a process's virtual address space. 10

microkernel A minimal operating system kernel that provides only the most essential services such as inter-process communication and basic scheduling, with other functionality moved to user space. 4

MMIO Memory-Mapped IO. 23–33

MMU Memory Management Unit. 10, 23, 24

mode switch The transition between user mode and kernel mode in an operating system, typically triggered by a system call or hardware interrupt.. 8, 10, 12, 15, 16, 20, 22, 23, 27, 31, 33

MVB Multifunction Vehicle Bus. 7, 15, 23–25, 33

NIC Network Interface Card. 20, 21

NVMe Non-Volatile Memory Express. 12, 21

page table A data structure used by the operating system to store the mapping between virtual addresses and physical memory addresses in a system that uses paging. 10

PCI Peripheral Component Interconnect. 22, 23

physical address The actual address in main memory (RAM) corresponding to a linear address after paging, used by hardware to access data. 10

PIS Passenger Information System. 7

PMD Poll Mode Driver. 21, 22

POSIX Portable Operating System Interface. 9

PTE Page Table Entry. 10

PTY Pseudo Terminal. 9, 26

QEMU Quick EMUlator. 24

RAM Random Access Memory. 26

Raspberry Pi A low-cost, credit-card-sized single-board computer developed by the Raspberry Pi Foundation.. 5, 26

ring buffer A fixed-size circular data structure that wraps around when the end is reached, commonly used in producer-consumer scenarios and buffering streams. 15

Rust A systems programming language focused on safety, speed, and concurrency, known for its memory safety without a garbage collector. 20, 33

SBC Single-Board Computer. 26

SPDK Storage Performance Development Kit. 15, 18, 21–24

SSD Solid State Drive. 12, 21

system call A mechanism for user programs to request services or resources from the operating system kernel.. 5, 6, 8, 9, 11, 15–17, 19, 20, 22, 26–28, 30, 31, 33

system call table A data structure within the Linux kernel that maps system call numbers to their corresponding kernel function implementations.. 9, 11

TCN Train Communication Network. 7

Televic A company specializing in innovative and reliable railway passenger information and communication systems.. 6, 7, 18, 24, 26, 34

TLB Translation Lookaside Buffer. 10

tracepoint A predefined location in code, typically in the kernel or user-space applications, where instrumentation can be inserted to collect runtime information for debugging or performance analysis. 19, 26, 27, 29, 33

UEFI Unified Extensible Firmware Interface. 24

UNIX UNiplexed Information Computing System. 9

user space The area of memory where user applications and processes run, isolated from the kernel. 4–8, 10, 15, 19, 22–26, 31, 33

vFIO Virtual Function IO. 15, 18, 23–25

vFS Virtual File System. 11

Appendix

Benchmarking Stats

File Logmodule Writes

```

Performance counter stats for 'build/syscall_benchmark':

    2,372,201,312      cycles
    2,250,131,293      instructions                     #    0.95  insn per cycle
       15,726,238      cache-misses
               25      context-switches
                6      syscalls:sys_enter_openat
        1,000,000      syscalls:sys_enter_write
                6      syscalls:sys_enter_close
        1,000,000      logmodule:write

      1.583646277 seconds time elapsed

      0.231864000 seconds user
      1.351212000 seconds sys
```

Figure 23: Systemcall access, writing 4b 1M times to Logmodule driver

```

Performance counter stats for 'build/syscall_benchmark':

    2,295,536,515      cycles
    2,251,129,497      instructions                     #    0.98  insn per cycle
        6,974,384      cache-misses
              155      context-switches
                6      syscalls:sys_enter_openat
        1,000,000      syscalls:sys_enter_write
                6      syscalls:sys_enter_close
        1,000,000      logmodule:write

      1.538086590 seconds time elapsed

      0.254733000 seconds user
      1.277649000 seconds sys
```

Figure 24: Systemcall access, writing 4kb 1M times to Logmodule driver

```

Performance counter stats for './build/dma_benchmark':

       10,637,259      cycles
       10,783,079      instructions                     #    1.01  insn per cycle
           30,301      cache-misses
                3      context-switches
                6      syscalls:sys_enter_openat
                0      syscalls:sys_enter_write
                6      syscalls:sys_enter_close
                0      logmodule:write

      0.009475802 seconds time elapsed

      0.004307000 seconds user
      0.004307000 seconds sys
```

Figure 25: DMA, writing 4b 1M times to Logmodule driver

```
Performance counter stats for './build/dma_benchmark':

       10,550,703      cycles
       10,761,280      instructions                     #    1.02  insn per cycle
           30,555      cache-misses
                2      context-switches
                6      syscalls:sys_enter_openat
                0      syscalls:sys_enter_write
                6      syscalls:sys_enter_close
                0      logmodule:write

      0.010459653 seconds time elapsed

      0.004299000 seconds user
      0.004299000 seconds sys
```

Figure 26: DMA, writing 4kb 1M times to Logmodule driver

```

 Performance counter stats for './build/uring_benchmark':

    9,803,326,630      cycles
    7,572,884,270      instructions                     #    0.77  insn per cycle
       60,982,970      cache-misses
           97,323      context-switches
                7      syscalls:sys_enter_openat
                0      syscalls:sys_enter_write
                8      syscalls:sys_enter_close
        1,000,000      logmodule:write

      2.856642305 seconds time elapsed

      0.664411000 seconds user
      7.237544000 seconds sys
```

Figure 27: IO_Uring access, writing 4b 1M times to Logmodule driver

```

Performance counter stats for './build/uring_benchmark':

     9,531,505,273      cycles
     7,399,208,137      instructions                     #    0.78  insn per cycle
        55,324,687      cache-misses
            90,528      context-switches
                 7      syscalls:sys_enter_openat
                 0      syscalls:sys_enter_write
                 8      syscalls:sys_enter_close
         1,000,000      logmodule:write

       2.514316225 seconds time elapsed

       0.670076000 seconds user
       6.947073000 seconds sys
```

Figure 28: IO_Uring access, writing 4kb 1M times to Logmodule driver

File System Writes

```

Performance counter stats for './build/uring_benchmark':

    11,898,953,592      cycles
     9,855,705,939      instructions                     #    0.83  insn per cycle
       146,037,595      cache-misses
           165,212      context-switches
                 7      syscalls:sys_enter_openat
                 0      syscalls:sys_enter_write
                 8      syscalls:sys_enter_close
                 0      logmodule:write

       3.313464772 seconds time elapsed

       0.611673000 seconds user
       9.063516000 seconds sys
```

Figure 29: IO_Uring access, writing 4kb 1M times to File System

USB Device Writes

```

Performance counter stats for 'build/syscall_benchmark':

    3,812,632,005      cycles
    3,548,118,317      instructions                     #    0.93  insn per cycle
        2,109,348      cache-misses
               35      context-switches
                6      syscalls:sys_enter_openat
        1,000,000      syscalls:sys_enter_write
                6      syscalls:sys_enter_close
                0      logmodule:write

      2.543988416 seconds time elapsed

      0.451892000 seconds user
      2.091504000 seconds sys
```

Figure 30: Systemcall access, writing 4b 500k times to USB Stick

```

Performance counter stats for 'build/syscall_benchmark':

    4,822,794,842      cycles
    4,922,603,906      instructions                     #    1.02  insn per cycle
       24,058,884      cache-misses
              182      context-switches
                6      syscalls:sys_enter_openat
        1,000,000      syscalls:sys_enter_write
                6      syscalls:sys_enter_close
                0      logmodule:write

      3.222870544 seconds time elapsed

      0.467026000 seconds user
      2.750270000 seconds sys
```

Figure 31: Systemcall access, writing 4kb 500k times to USB Stick

```

Performance counter stats for './build/vfio_benchmark':

       10,561,813      cycles
       10,762,628      instructions                     #    1.02  insn per cycle
           30,623      cache-misses
                2      context-switches
                6      syscalls:sys_enter_openat
                0      syscalls:sys_enter_write
                6      syscalls:sys_enter_close
                0      logmodule:write

      0.010772506 seconds time elapsed

      0.004274000 seconds user
      0.004274000 seconds sys
```

Figure 32: DMA, writing 4b to 500k times USB Stick

```

Performance counter stats for './build/vfio_benchmark':

       10,736,817      cycles
       10,798,699      instructions                     #    1.01  insn per cycle
           30,893      cache-misses
                9      context-switches
                6      syscalls:sys_enter_openat
                0      syscalls:sys_enter_write
                6      syscalls:sys_enter_close
                0      logmodule:write

      0.010811824 seconds time elapsed

      0.004366000 seconds user
      0.004366000 seconds sys
```

Figure 33: DMA, writing 4kb 500k times to USB Stick

```

Performance counter stats for './build/uring_benchmark':

   10,701,390,849      cycles
    8,045,024,339      instructions                     #    0.75  insn per cycle
       73,784,239      cache-misses
          113,283      context-switches
                7      syscalls:sys_enter_openat
                0      syscalls:sys_enter_write
                8      syscalls:sys_enter_close
                0      logmodule:write

      3.083523685 seconds time elapsed

      0.763746000 seconds user
      7.849827000 seconds sys
```

Figure 34: IO_Uring access, writing 4kb 500k times to USB Stick

```

Performance counter stats for './build/uring_benchmark':

   14,205,213,058      cycles
   11,837,856,905      instructions                     #    0.83  insn per cycle
       83,216,466      cache-misses
          154,374      context-switches
                7      syscalls:sys_enter_openat
                0      syscalls:sys_enter_write
                8      syscalls:sys_enter_close
                0      logmodule:write

      3.941801096 seconds time elapsed

      0.676058000 seconds user
     10.620009000 seconds sys
```

Figure 35: IO_Uring access, writing 4kb 500k times to USB Stick

Misc

```

Performance counter stats for './build/uring_benchmark':

   75,897,674,295      cycles
   64,999,334,720      instructions                     #    0.86  insn per cycle
      584,981,560      cache-misses
        2,000,516      context-switches
                7      syscalls:sys_enter_openat
                0      syscalls:sys_enter_write
                8      syscalls:sys_enter_close
        1,000,000      logmodule:write

     31.614919151 seconds time elapsed

      1.286808000 seconds user
     62.596308000 seconds sys
```

Figure 36: IO_Uring used in blocking loop, writing 4b 1M times to Logmodule driver

Graphs and Visualizations

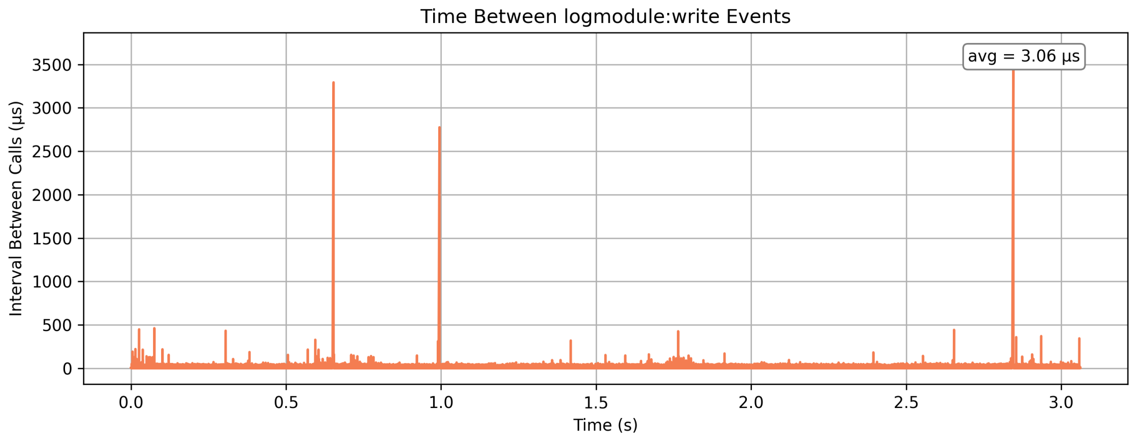


Figure 37: Syscall time between writes 1M/4b

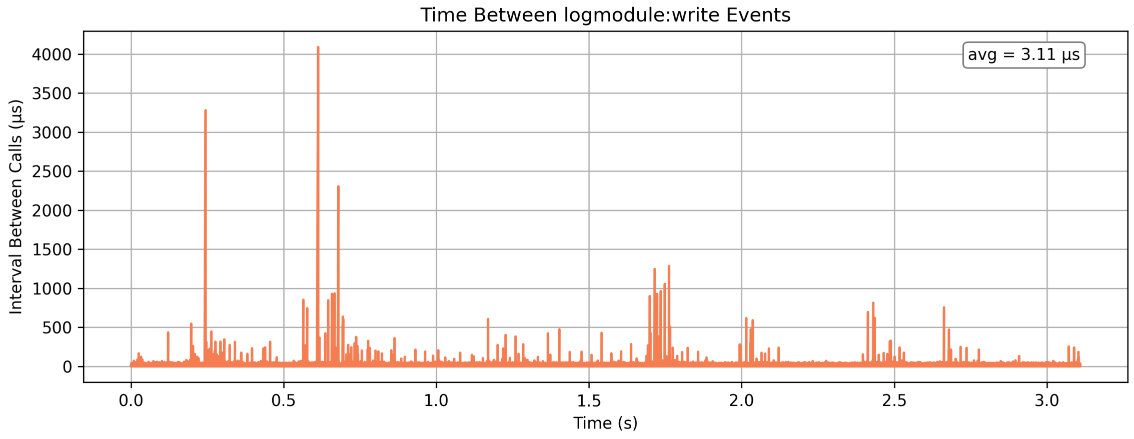


Figure 38: Syscall time between writes 1M/4kb

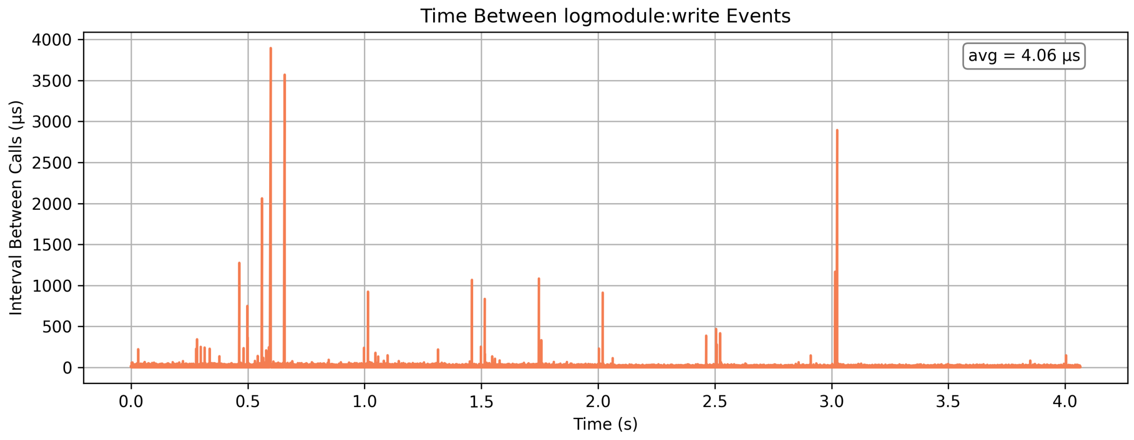


Figure 39: IO_Uring time between writes 1M/4b

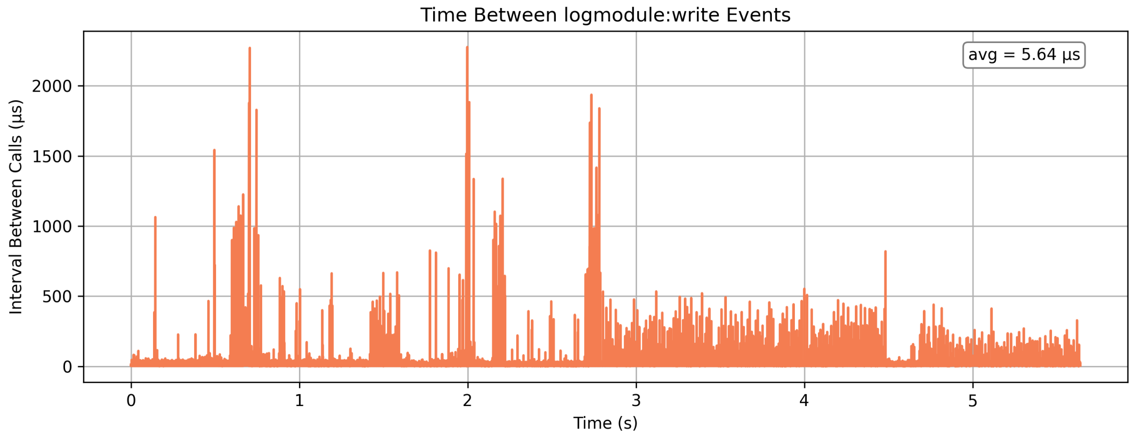


Figure 40: IO_Uring time between writes 1M/4kb

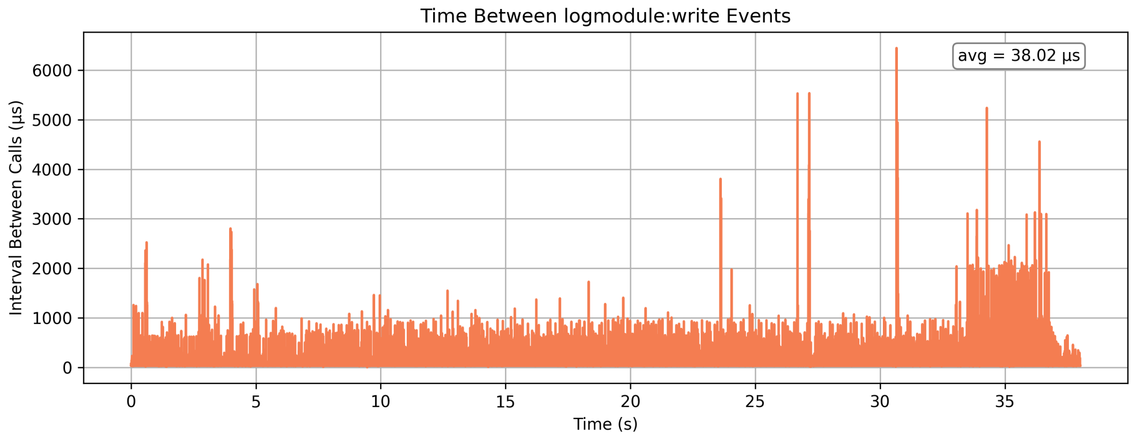


Figure 41: IO_Uring blocking call cycle

Performance Overview

Run	User Time	Sys Time	Total Time	User %	Sys %
systemcall 4b	0.232	1.351	1.583	14.6%	85.4%
systemcall 4kb	0.255	1.278	1.532	16.6%	83.4%
io_uring 4b	0.664	7.238	7.902	8.4%	91.6%
io_uring 4kb	0.670	6.947	7.617	8.8%	91.2%
dma 4b	0.004	0.004	0.009	50.0%	50.0%
dma 4kb	0.004	0.004	0.009	50.0%	50.0%

Figure 42: Performance benchmarking overview, one million writes to system endpoint

Run	User Time	Sys Time	Total Time	User %	Sys %
systemcall 4b to USB	0.452	2.092	2.543	17.8%	82.2%
systemcall 4kb to USB	0.467	2.750	3.217	14.5%	85.5%
io_uring 4b to USB	0.764	7.850	8.614	8.9%	91.1%
io_uring 4kb to USB	0.676	10.620	11.296	6.0%	94.0%
dma 4b to USB	0.004	0.004	0.009	50.0%	50.0%
dma 4kb to USB	0.004	0.004	0.009	50.0%	50.0%

Figure 43: Performance benchmarking overview, one million writes to hardware endpoint

Graphs

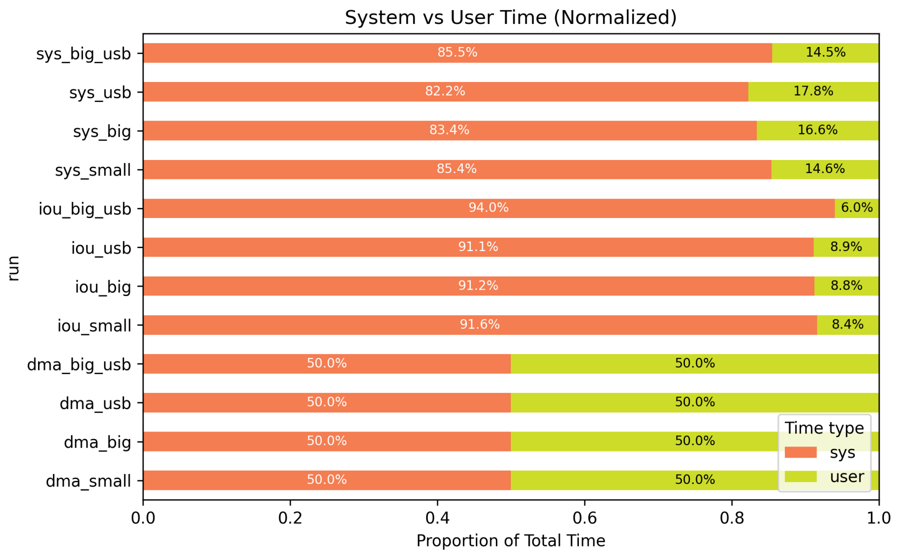


Figure 44: Normalized proportions for writes

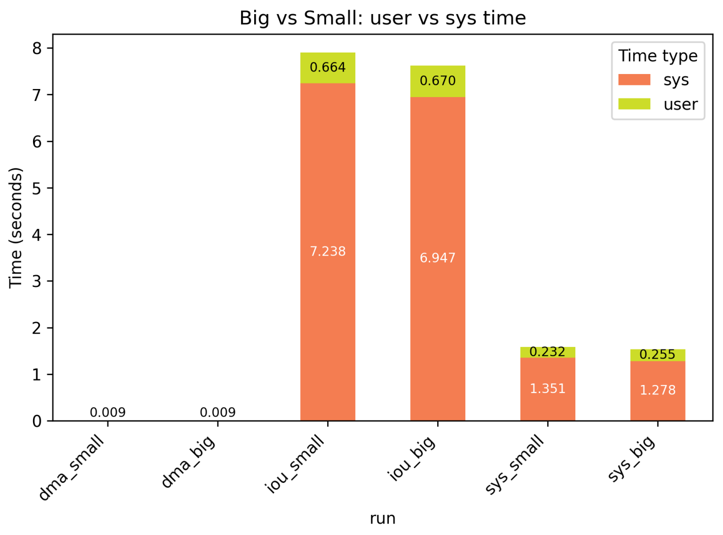


Figure 45: Performance Benchmarking: Total CPU time SYS write

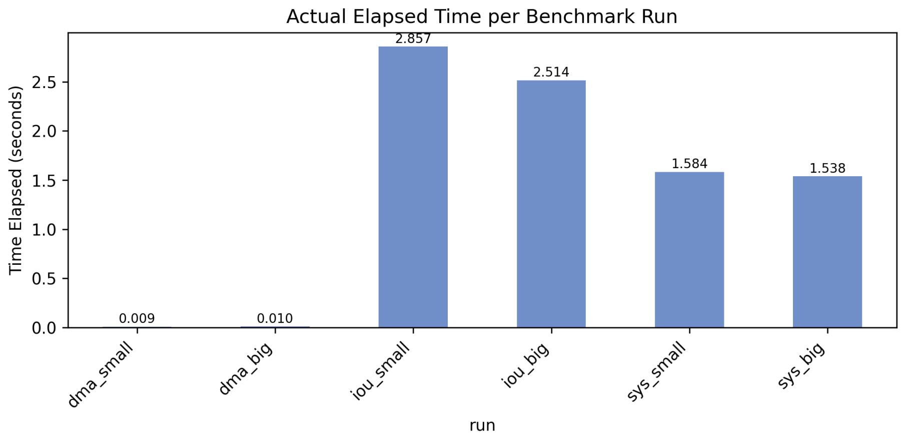


Figure 46: Performance Benchmarking: Total Time taken SYS write

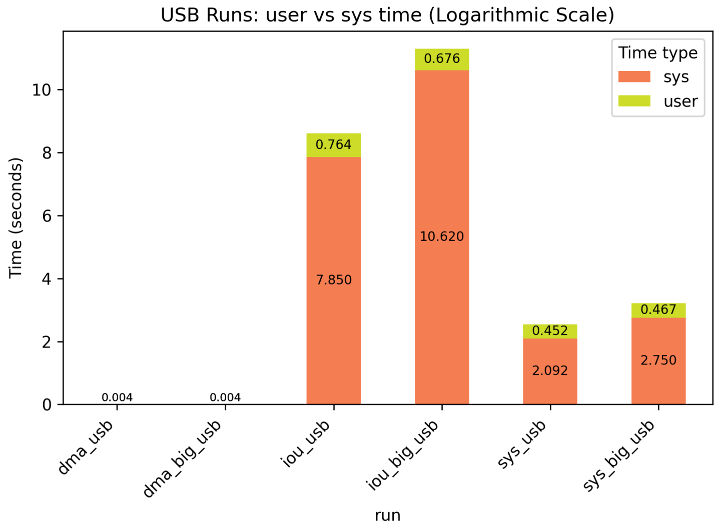


Figure 47: Performance Benchmarking: Total CPU time USB write

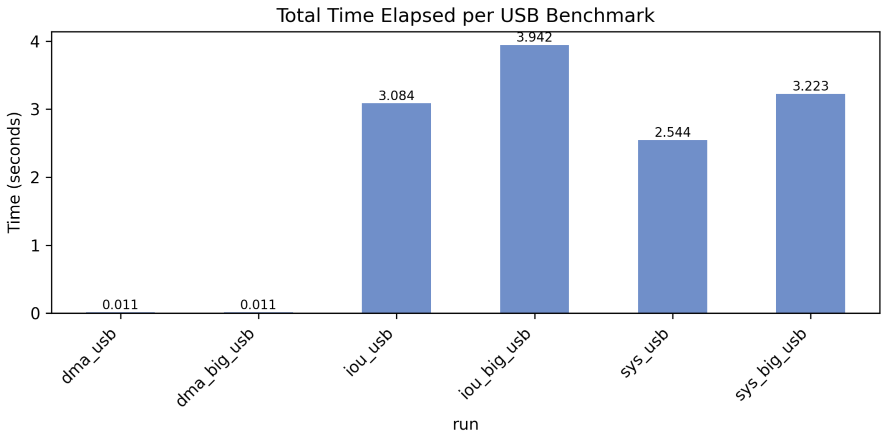


Figure 48: Performance Benchmarking: Total time taken USB Write

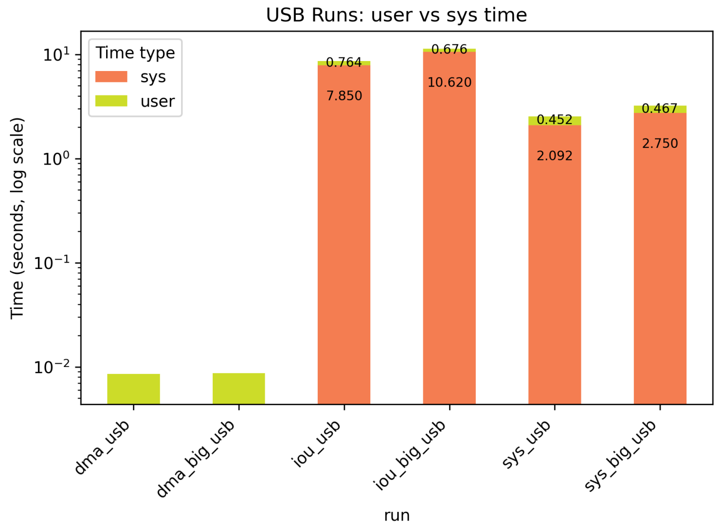


Figure 49: Performance Benchmarking: Total CPU time USB write (logarithmic)