Quantifying the Overheads of the Modern Linux I/O Stack

Luke Logan, Anthony Kougkas, and Xian-He Sun
Department of Computer Science, Illinois Institute of Technology, Chicago, IL
llogan@hawk.iit.edu, akougkas@iit.edu, sun@iit.edu

***Index Terms*—I/O Bottleneck, Filesystems, Linux**

I. Extended Abstract

Linux is the foundation of 9 of the top 10 public clouds [1] and all Top500 supercomputers [2]. Several distributed storage services such as Object Stores, Parallel File Systems, and Databases (e.g., OrangeFS [3]) largely rely on the Linux I/O stack for their storage needs. They store data using the UNIX file representation and access these files using the POSIX interface that Linux provides. Thus, the performance of the Linux I/O stack is critical to the performance of these applications as a whole. However, recent research has shown that the Linux I/O stack introduces multiple overheads that significantly reduce and randomize the performance of I/O operations. Cao et al. (2017) [4] found that, for multiple workloads and filesystems, running the same exact workload multiple times on the same filesystem configuration resulted in performance variations of as much as 40% between runs and that the performance variation was especially chaotic when the underlying storage device was an HDD, as opposed to an SSD. It was found that lazy block allocation and randomization of the block layout on disk were major contributors to this variation. Furthermore, the Linux I/O stack is programmed for slower storage devices and is shown to cause significant performance overheads for fast storage like NVMe and SSD [5]. For this reason, multiple services that bypass the Linux I/O stack for these particular devices have been proposed [5], [6]. While some research has been conducted into identifying the source of performance concerns and some tools have been built to bypass these overheads in certain cases, not much research has been conducted into quantifying the software overheads in the Linux I/O stack. *This is important to understanding whether or not the I/O stack needs to be completely redesigned for HPC and Cloud systems.*

In this research, we quantify the software overheads in the Linux I/O stack by tracing the POSIX `read()`/`write()` system calls on various storage devices and filesystems. By comparing the amount of time spent in software versus the amount of time spent in performing I/O, we can gain insight on how much overhead the Linux I/O stack produces and explore solutions that can mitigate the overheads.

II. The Linux I/O Stack

In the Linux I/O stack (up to version 5.8, shown in Figure 2 of the poster), the user reserves a chunk of virtual memory using an allocator function (e.g., `malloc()`) and then passes this virtual address, along with a length and file descriptor, to POSIX I/O syscalls such as `read()` or `write()`. When this happens, the Virtual File System (VFS) Layer takes over.

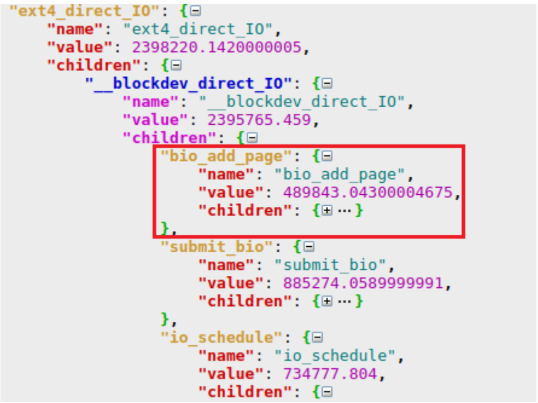
The VFS Layer is mainly responsible for updating file metadata and for journaling (if applicable). File metadata is stored in a `struct inode`, which contains information such as timestamps, the owner of the file, and the mapping between file offsets and disk blocks. The set of disk blocks to be accessed in the I/O request are identified using this inode. After the file metadata has been updated, the set of disk blocks and the user's buffer are passed either to the Page Cache or to the Direct I/O (DIO) Layer. At this point, Block I/O (BIO) requests (of type `struct bio`) will be constructed that will either flush pages to the disk or read pages from the disk. A single BIO associates an array of pages in RAM with a contiguous set of disk blocks to either be read or modified. If the Page Cache is used, BIOs will be constructed using pages from the cache. Otherwise, the user's buffer is converted into an array of pages (using `get_user_pages()`) and then used in the BIOs. The BIOs will then be passed to the BIO Layer using `submit_bio()`.

The BIO Layer [7] is mainly responsible for plugging/merging/splitting BIOs and for passing BIOs to the Request Layer. Plugging is a mechanism temporarily prevents BIOs from being scheduled in the Request Layer so that they can be merged to form larger, contiguous BIOs and reduce the number of small requests. Splitting is used to divide BIOs that are too large for the underlying device to handle. After the plug is finished, the BIOs are converted into "requests" (of type `struct request`) and passed to the Request Layer in `blk_finish_plug()` or `io_schedule()`.

The Request Layer [8] is responsible for scheduling requests and passing requests to the device drivers. There are two types of schedulers: Single-Queue (SQ) and Multi-Queue (MQ). SQ is designed for devices that can't handle concurrent I/O requests such as HDD. MQ is designed for both kinds of devices. Either way, the requests in the queues get ordered according to some policy and then passed to the device drivers. The device drivers are ultimately responsible for interacting with the device and handling interrupts.

III. Quantifying Overheads

Approach: To capture the impact of the software overheads in the Linux I/O stack, we clear the OS page cache



```
"ext4_direct_IO": {⊟
    "name": "ext4_direct_IO",
    "value": 2398220.1420000005,
    "children": {⊟
        "__blockdev_direct_IO": {⊟
            "name": "__blockdev_direct_IO",
            "value": 2395765.459,
            "children": {⊟
                "bio_add_page": {⊟
                    "name": "bio_add_page",
                    "value": 489843.04300004675,
                    "children": {⊞ ⋯ }
                },
                "submit_bio": {⊟
                    "name": "submit_bio",
                    "value": 885274.0589999991,
                    "children": {⊞ ⋯ }
                },
                "io_schedule": {⊟
                    "name": "io_schedule",
                    "value": 734777.804,
                    "children": {⊟
```

Fig. 1. Function graph of 100MB `write()` to SSD where `value` is in μs

and then perform either sequential reads or writes of 1GB for various block sizes (4KB, 64KB, 1MB, 10MB, 100MB) using the read()/write() system calls on files opened with the O_DIRECT flag (in order to bypass the Linux Page Cache). We bypass the page cache so that we can compare the amount of time spent in software and the maximum amount of time spent in I/O. We also vary whether or not the file was allocated in the filesystem. Furthermore, we perform this experiment on various filesystems (EXT4 and XFS) and storage devices (SAS HDD and SATA SSD). We used `trace-cmd` [9] in order to trace the system calls and produce a function graph of them. Lastly, we implemented a kernel module that demonstrates the performance improvement of removing the overheads in the VFS and DIO layers when performing I/O by timing the period between the BIO submission and the completion of the I/O.

Testbed: We performed our tests on Chameleon Cloud [10] using a Compute Haswell Node and a Compute Skylake Node. The Haswell Node contains a single 250GB, 7200 RPM SAS HDD and a 2.3GHz 12-core/24-thread Intel Xeon CPU. The Skylake Node contains a single 240GB SATA SSD and a 2.6GHz 12-core/24-thread Intel Xeon CPU. We ran our experiments on Ubuntu 18.04 with Linux 4.15.0-101-generic.

Results: Due to the limited space of this submission, we will focus on the results obtained from sequentially writing 100MB of data to a preallocated file located on an SSD with an EXT4 filesystem. From Figure 1, we see that 20% of the time was spent in `bio_add_page()` (489,843 of the 2,398,220 μs). We also found an additional 6% was spent in splitting and merging BIOs (not shown in the figure). Thus, at least **26%** of the time consumed by `write()` is spent in software operations required by the Linux I/O Stack.

IV. Conclusion and Next Steps

From our experiments, we see that a large fraction of time spent in I/O is spent in constructing BIOs and splitting/merging BIOs. For these reasons, we believe that these software overheads are significant. Thus, our next step is to devise a way to bypass these overheads. There are three general categories of methods for bypassing the overheads of the current Linux I/O stack: 1) *Other Linux APIs*: Linux includes other interfaces for performing I/O outside of POSIX, such as Memory-Mapped I/O (MMIO) and Direct Access (DAX) to files. They map blocks of the storage device into the user's address space. Instead of using explicit system calls, page faults are used to perform I/O. However, MMIO/DAX can't map entire storage devices into RAM and they incur other overheads such as creating Page Table Entries. 2) *Linux Kernel Modules*: Linux Kernel Modules can be used to expose the underlying device drivers or even communicate with the hardware itself. However, kernel modules can be hard to program and maintain, especially when kernel APIs change from version to version. 3) *Alternate (non-Linux) Kernel Architectures*: Servers could switch from Linux to a completely different kernel. Linux is a monolithic kernel, and includes a lot of functionality, such as filesystems and device drivers, in order to make it portable to a large and diverse population. However, alternate kernel architectures such as Microkernels and Unikernels exist that allow users to build highly specialized operating systems tailored for a certain purpose. No I/O stack is provided by these types of kernels. However, these kernels are typically only used with the help of Hypervisors due to limited driver support.

In summary, we showed the potential to boost the performance of a storage server by quantifying the software overheads of the existing Linux I/O stack and proposed several ways to bypass these overheads. Given this, we plan to design and develop *a new, high-performance, lightweight, and robust storage software stack* for data-intensive computing and its new data representations.

References

[1] RedHat, "The state of linux in the public cloud for enterprises." RedHat, 2017. [Online]. Available: https://www.redhat.com/en/resources/state-of-linux-in-public-cloud-for-enterprises

[2] Top500.org. Top500, June 2020. [Online]. Available: https://www.top500.org/lists/top500/2020/06/

[3] "Orangefs." OrangeFS, 2020. [Online]. Available: http://www.orangefs.org/

[4] Z. Cao, V. Tarasov, H. P. Raman, D. Hildebrand, and E. Zadok, "On the performance variation in modern storage stacks," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, Feb. 2017, pp. 329–344. [Online]. Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/cao

[5] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "Spdk: A development kit to build high performance storage applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017, pp. 154–161.

[6] I. Zhang, J. Liu, A. Austin, M. L. Roberts, and A. Badam, "I'm not dead yet! the role of the operating system in a kernel-bypass era," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 73–80. [Online]. Available: https://doi.org/10.1145/3317550.3321422

[7] N. Brown, "A block layer introduction part 1: the bio layer." LWN, 2017. [Online]. Available: https://lwn.net/Articles/736534/

[8] ——, "Block layer introduction part 2: the request layer." LWN, 2017. [Online]. Available: https://lwn.net/Articles/738449/

[9] S. Rostedt, "trace-cmd." RedHat, 2010. [Online]. Available: https://man7.org/linux/man-pages/man1/trace-cmd.1.html

[10] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the chameleon testbed," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.