Quantifying the Overheads of the Modern Linux I/O Stack

Luke Logan Anthony Kougkas Xian-He Sun
llogan@hawk.iit.edu, akougkas@iit.edu, and sun@iit.edu

SCALABLE COMPUTING SOFTWARE LABORATORY

ILLINOIS INSTITUTE OF TECHNOLOGY

Abstract

Linux is the foundation of 9 of the top 10 public clouds [5] and all Top500 supercomputers [7]. Several distributed storage services such as Object Stores, Parallel File Systems, and Databases (e.g., OrangeFS [1]) 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 requests [2, 9, 8]. 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 I/O, we can gain insight on how much overhead the Linux I/O stack produces and propose solutions that can mitigate the overheads.

Testbed

	Haswell	Skylake
OS	Ubuntu 18.04	Ubuntu 18.04
Linux	4.15.0-101-generic	4.15.0-101-generic
CPU (cores)	12	12
CPU (threads)	24	24
Storage Type	SAS HDD	SATA SSD
Capacity	250GB	240GB

Figure 1: Chameleon Cloud [3]

Methodology

- Preallocate file of 1GB in filesystem
- Clear OS page cache before every test
- Use O_DIRECT flag to bypass page cache
- Sequential, synchronous I/O using POSIX `read()/write()`
- Vary I/O request size, filesystem, and storage
- Repeat each test at least 100 times
- Use trace-cmd [6] to find sources of overhead

The Linux I/O Stack

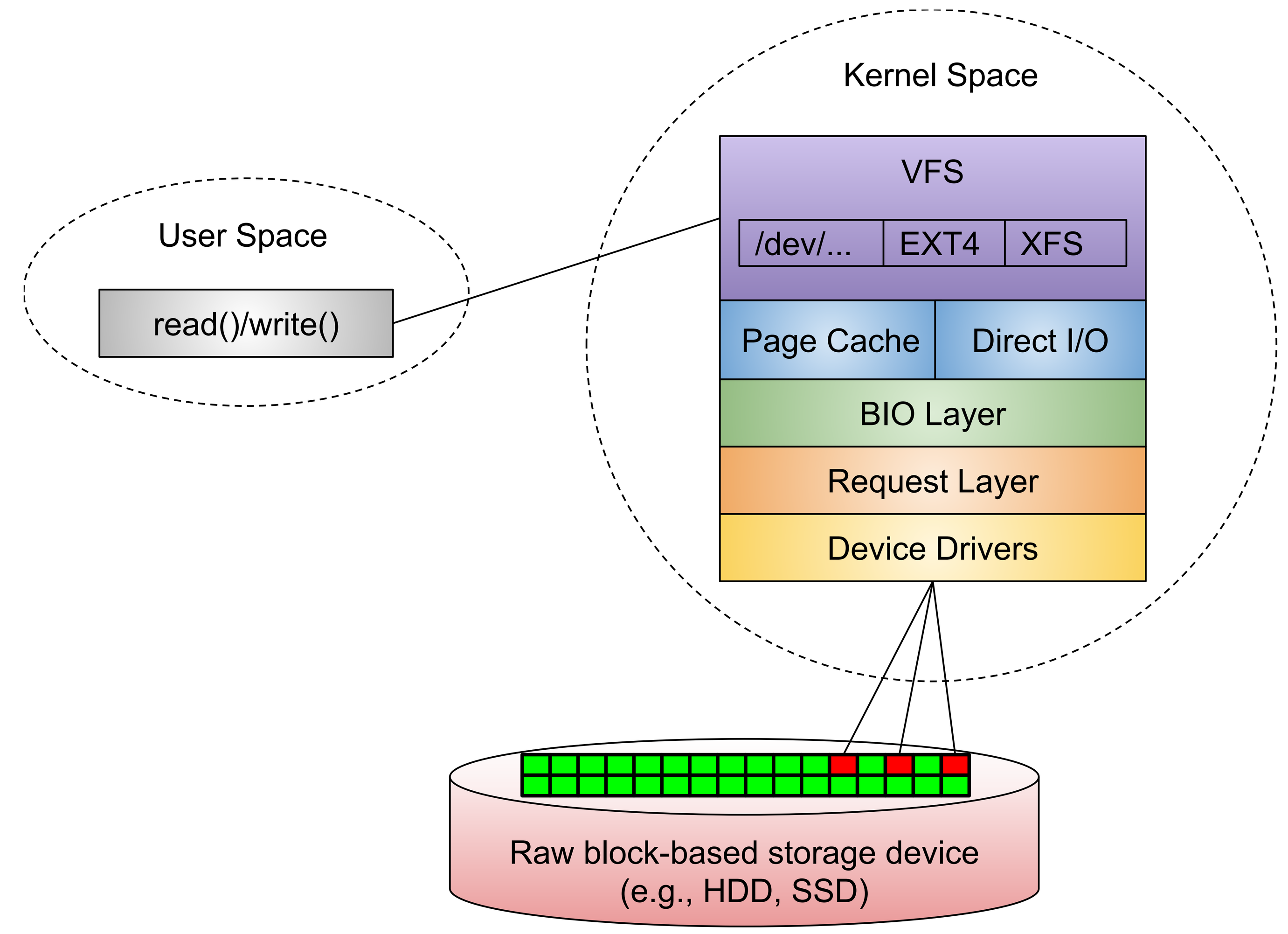

Figure 2: Linux I/O Stack

- **User Space:** Reserve a chunk of virtual memory using an allocator function (e.g., `malloc()`) and pass this virtual address, along with a length and file descriptor, to POSIX I/O syscalls such as `read()` or `write()`
- **VFS Layer:** Update file metadata and perform journaling (if applicable). Discover the set of disk blocks to be used in the I/O request and pass this information, along with the user's buffer and length, to either the Page Cache or Direct I/O (DIO) Layer.
- **Page Cache:** Construct/submit Block I/O requests (BIOs) that associate pages in the cache with disk blocks. I/O does not happen directly with the user's buffer; data must be copied between the cache and the user's buffer.
- **DIO Layer:** Convert the user's buffer into pages and then construct/submit BIOs that associate those pages with disk blocks.
- **BIO Layer:** Plug/merge/split BIOs and convert BIOs into requests. Plug waits for additional BIOs. Merge combines contiguous BIOs into one BIO. Split divides BIOs that are too large for underlying hardware to handle.
- **Request Layer:** Schedule/order requests and pass requests down to the device drivers.
- **Device Drivers:** Send commands and handle interrupts.

Bypassing the Linux I/O Stack

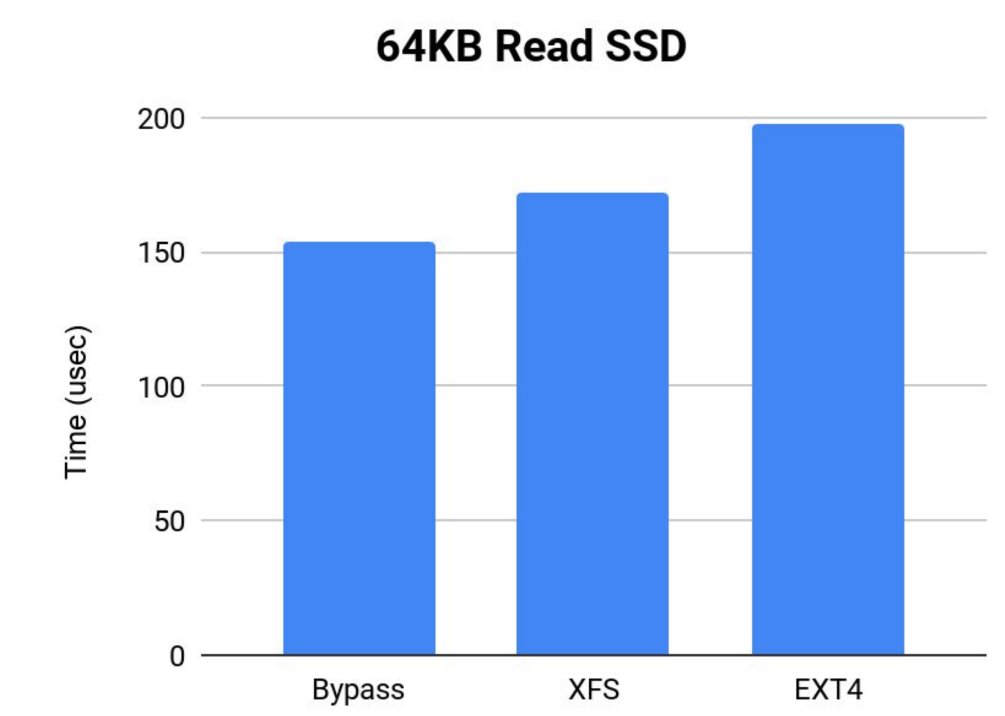

Figure 5: Sequential Reads of 64KB from XFS and EXT4 SSDs

- We built a kernel module [4] that ignores the cost of constructing BIOs
- Sequential read of 6.4MB in blocks of size 64KB on Skylake
- **10% faster than XFS**
- **20% faster than EXT4**

Profiling the Linux I/O Stack

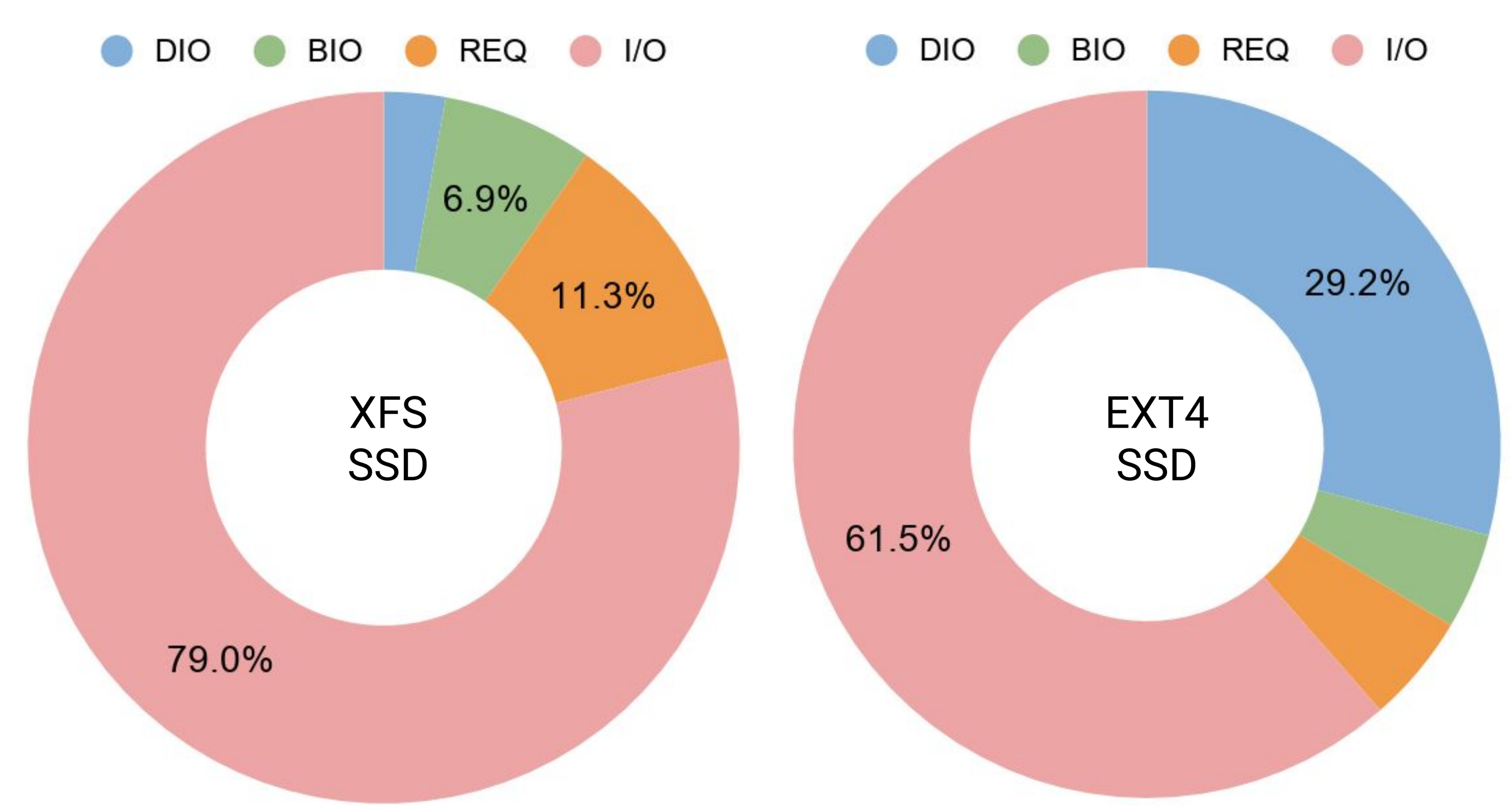

Figure 3: Sequential Writes of 10MB from XFS and EXT4 SSDs

- 2.8% of time spent creating BIOs
- 6.9% of time spent splitting/merging/plugging BIOs
- 11.3% of time spent reordering/queuing requests
- **21% of time spent in software**
- SSDs have fast random access
- Filesystems do not leverage the architecture of SSDs

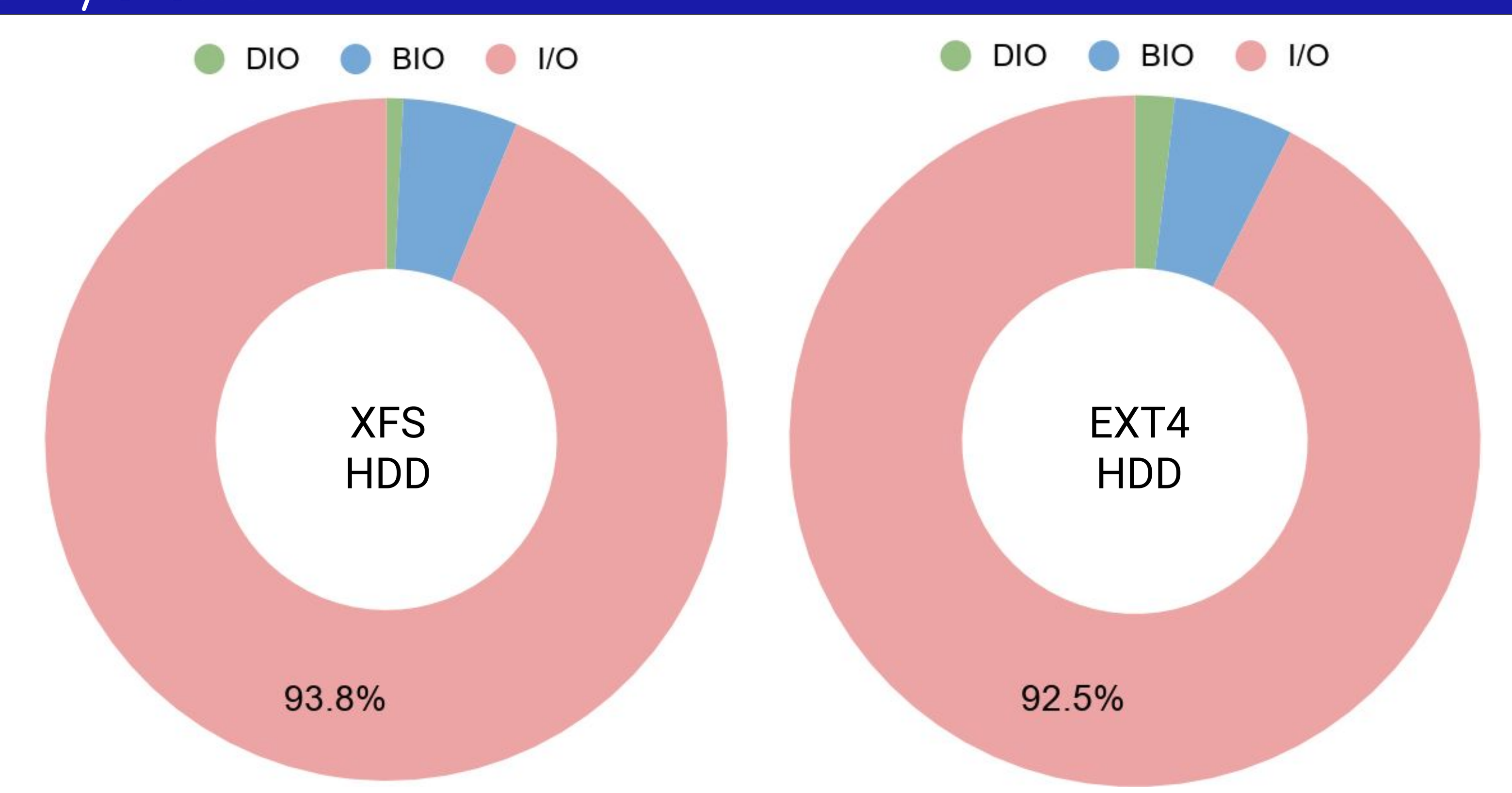

Figure 4: Sequential Reads of 10MB from EXT4 and XFS HDD

- .8% of time spent creating BIOs
- 5.4% of time spent splitting/merging/plugging BIOs
- **6.2% of time spent in software**
- A significant amount of time is spent merely constructing/splitting/merging BIOs
- This is true across storage architectures

Conclusion

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] (2020). Orangefs. OrangeFS. http://www.orangefs.org/.

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

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

[4] Logan, L. (2020). https://github.com/lukemartinlogan/linux-bio-km.

[5] RedHat (2017). The state of linux in the public cloud for enterprises. RedHat. https://www.redhat.com/en/resources/state-of-linux-in-public-cloud-for-enterprises.

[6] Rostedt, S. (2010). trace-cmd. RedHat. https://man7.org/linux/man-pages/man1/trace-cmd.1.html.

[7] Top500.org (2020). Top500. https://www.top500.org/lists/top500/2020/06/.

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

[9] Zhang, I., Liu, J., Austin, A., Roberts, M. L., and Badam, A. (2019). 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*, HotOS '19, page 73–80, New York, NY, USA. Association for Computing Machinery. https://doi.org/10.1145/3317550.3321422.