Keeping It Real: Why HPC Data Services Don't Achieve I/O Microbenchmark Performance

Philip Carns,* Kevin Harms,* Bradley W. Settlemyer,† Brian Atkinson,† Robert B. Ross*
*Argonne National Laboratory, †Los Alamos National Laboratory
*carns@mcs.anl.gov, harms@alcf.anl.gov, rross@mcs.anl.gov, †bws@lanl.gov, batkinson@lanl.gov

***Abstract*—HPC storage software developers rely on benchmarks as reference points for performance evaluation. Low-level synthetic microbenchmarks are particularly valuable for isolating performance bottlenecks in complex systems and identifying optimization opportunities.**

The use of low-level microbenchmarks also entails risk, however, especially if the benchmark behavior does not reflect the nuances of production data services or applications. In those cases, microbenchmark measurements can lead to unrealistic expectations or misdiagnosis of performance problems. Neither benchmark creators nor software developers are necessarily at fault in this scenario, however. The underlying problem is more often a subtle disconnect between the objective of the benchmark and the objective of the developer.

In this paper we investigate examples of discrepancies between microbenchmark behavior and software developer expectations. Our goal is to draw attention to these pitfalls and initiate a discussion within the community about how to improve the state of the practice in performance engineering for HPC data services.

I. Introduction and Background

Benchmarks are a crucial tool for understanding the capabilities of hardware and software resources employed in storage systems. In the ideal case, benchmarks are perfect proxies for application workloads and can accurately characterize end-to-end storage system performance. As storage systems become more complex, however, it becomes necessary to modularize storage software construction [1], [2]. HPC data services are also increasingly tailored to more specific use cases such as highly-selective scientific queries [3] or scientific machine learning [4]. These factors make it difficult to to fully understand underlying performance issues at scale.

Developers often attempt to isolate the performance of storage subsystems [5]–[7] to make performance engineering more tractable. Microbenchmarks are a natural choice to act as reference points for the performance of individual subsystems, but many popular microbenchmarks are designed to extract maximum hardware performance in their default configuration. In doing so, they employ techniques that are difficult to apply in storage use cases, omit operating system or user-space factors, or simply measure workloads that do not correlate well with common usage patterns. The discrepancy between the hardware access patterns in popular benchmarks and hardware access patterns in HPC data services can be surprisingly difficult to recognize without a deep understanding of low-level system architecture.

Performance engineering for storage systems development is an inherently detail-oriented task, but even with decades of storage system performance tuning experience, the authors were surprised by the low-level behavior of popular microbenchmarks. In this work we highlight common incorrect assumptions about several popular microbenchmarks, identify how the microbenchmark measurements differ from expectations, and initiate a discussion about how to use microbenchmarks effectively for performance engineering.

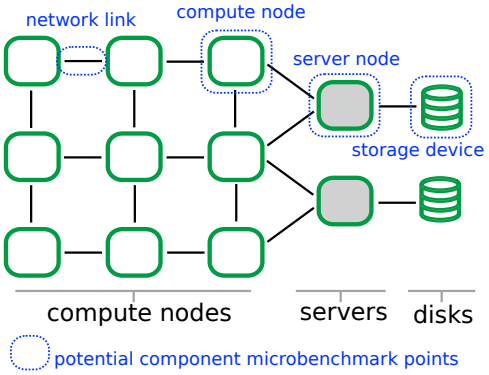

Fig. 1: HPC storage systems have many constituent hardware components, as illustrated in this conceptual diagram. Auxiliary resources such as archival and remote transfer systems are omitted for clarity.

The HPC storage community is well-positioned to provide leadership in storage system component performance benchmarking. Within many of our systems, performance is the primary concern – ranking ahead of conventional distributed systems concerns such as consistency, availability, or partition tolerance. This paper is born out of our initial frustration in leveraging roofline models [8] for I/O performance analysis, and we present this data to help practitioners avoid tripping over the same obstacles. The remainder of this section summarizes background and provides additional context for our study. Section II explores empirical case studies on two leading HPC platforms. Section III summarizes our findings and discusses their implications for HPC storage researchers and practitioners engaged in performance engineering activities.

A. HPC storage systems

HPC storage systems consist of compute nodes that issue storage requests, a network fabric that transfers data, service nodes that aggregate devices, and individual devices that store data. A conceptual diagram is shown in Figure 1.

Application-level performance measurements in this environment inevitably lead to a variety of follow-up questions: Did the perceived application performance match platform expectations? Which constituent component was the bottleneck? Can performance be improved? Answering these questions requires reference points (empirical measurements, models, or subject matter experience) to aid in contextualizing and interpreting performance. The dashed boxes in Figure 1 highlight key components that can be benchmarked, in aggregate or in isolation, to help provide these reference points.

B. Benchmarking techniques

A vast array of I/O benchmarks have been created for various purposes including stress testing, platform comparison, performance tuning, and advertising. Synthetic benchmarks emphasize generality and flexibility, while application proxy and trace-based benchmarks emphasize workload reproduction accuracy. Either type of benchmark may be appropriate during performance engineering whether performing roofline model analysis or some other type of analysis.

We focus on synthetic benchmarks in this study because they are the most straightforward type of benchmark to use for component-level measurements (i.e., *microbenchmarking*). Traces and proxies are difficult to obtain at the component level, especially on storage systems that support a diverse, multi-tenant portfolio of applications at an HPC facility.

II. Case studies

We present network, CPU, and storage case studies that focus on illustrative examples rather than exhaustive evaluation. Version information for all relevant benchmarks and support libraries can be found in the appendix.

We used two platforms for these experiments: the Summit supercomputer[1] operated by the Oak Ridge Leadership Computing Facility and the Theta supercomputer[2] operated by the Argonne Leadership Computing Facility. Summit consists of 4,608 compute nodes connected via EDR InfiniBand. Each Summit node contains 2 IBM Power9 CPUs, 6 NVidia Volta GPUs, 512 GiB of RAM, and a locally attached 1.6 TiB NVMe drive. Theta consists of 4,392 compute nodes connected by an Aries dragonfly network. Each node contains 64 Intel Knights Landing compute cores, 192 GiB of RAM, and a locally attached 128 GiB NVMe drive.

Variability is an important factor in storage performance that can also lead to misinterpretation if not treated properly [9], [10]. We recorded 50 sustained performance measurement samples for each benchmark configuration in an attempt to account for this. All 50 samples were collected on the same nodes while round-robin alternating between benchmark configurations. Violin plots are used to illustrate probability distribution, while horizontal lines are used to indicate minimum, median, and maximum sample values.

[1]https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

[2]https://www.alcf.anl.gov/support-center/theta

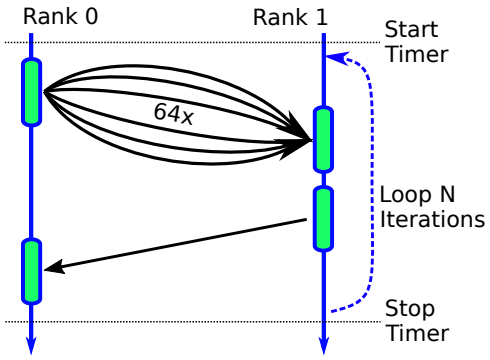

Fig. 2: Illustration of the `osu_bw` microbenchmark communication protocol: 64 concurrent asynchronous MPI messages are issued, each process blocks until all 64 are complete, an acknowledgement is sent in the opposite direction, and the sequence is repeated. All 64 concurrent messages are sent from (or received into) the same memory offset for the duration of the benchmark.

A. Network: memory access case study

Network performance is a crucial factor in distributed data service performance and is therefore a frequent benchmarking target for HPC data service developers. A variety of methods exist for measuring network performance, but MPI implementations and their benchmarks are a natural choice on HPC systems. MPI is portable, widely available, and optimized for asynchronous operation, efficient user-space access, and low latency. However, MPI libraries and HPC data services operate under different assumptions with respect to workloads, failure handling, communication symmetry, operating system interaction, and other factors. Memory access strategy is a notable example: does the benchmark repeatedly transfer "hot" memory, or does it iterate over a large memory region? The former obtains higher performance because it minimizes the overhead of memory registration and address translation. The latter is more representative of a large data service transfer, and the performance discrepancy between the two can be significant.

An example of hot memory reuse in an MPI benchmark is illustrated in Figure 2 which shows the communication pattern employed by the `osu_bw` microbenchmark from the OSU benchmark suite[3]. It exhibits two behaviors that differ from common practice in HPC data services. The first is that concurrency is achieved in 64-operation bursts rather than continuously issuing new operations as old ones complete. The second is that memory regions are reused repeatedly. All sends originate from a single memory offset on rank 0 and all receives arrive at a single memory offset on rank 1. Those memory offsets are also used in subsequent loop iterations.

The impact of this communication pattern is shown in Figure 3. The baseline measurement shows the performance of the standard `osu_bw` communication pattern as illustrated in Figure 2. The "patched" measurement shows the performance

[3]http://mvapich.cse.ohio-state.edu/benchmarks/

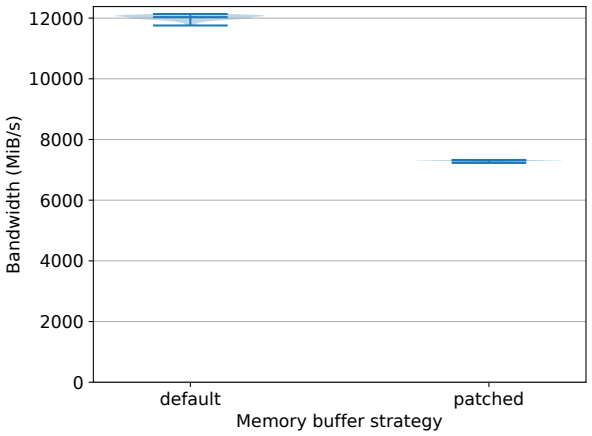

Fig. 3: Point-to-point internode MPI bandwidth reported by `osu_bw` with a 32 KiB message size on Summit.

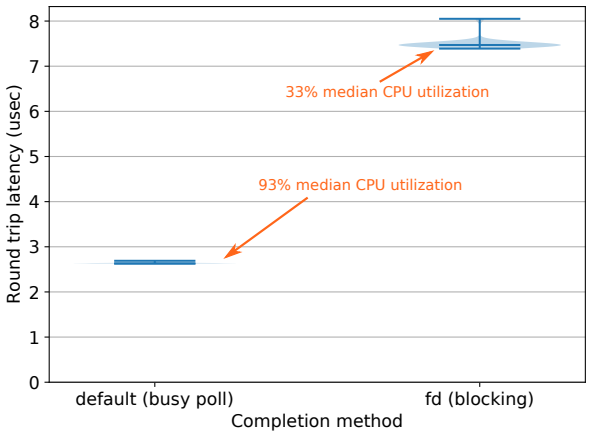

Fig. 4: Point-to-point internode round-trip latency as reported by `fi_msg_pingpong` with a 100-byte message size on Summit. The default mode uses busy polling, while the fd blocking mode uses the `-c fd` command line option.

of `osu_bw` with the following modification: each process allocates a 1 GiB communication buffer and then sends from or receives into regions in that buffer in a round-robin fashion, incrementing the offset by X bytes for each message. The modified benchmark exhibits a nearly 40% performance penalty, even though the two configurations are nominally issuing equivalent network operations. A user-space HPC data service would require careful optimization (for example, by copying data through reusable, aligned transfer buffers) to approach the idealized messaging performance reported by the baseline MPI benchmark configuration.

B. Network: completion method case study

Network latency is also an important element in distributed data service performance. It places a lower bound on response time, which is crucial for metadata operations, sequential I/O operations, and interactive use. One of the most influential factors in network latency is not technically a network phenomenon at all: it is the time required for a process to receive notification of network message completion. The highest performance method for retrieving network notifications in user space is for user processes to continuously poll the network device. This technique avoids significant interrupt, signaling, and context switching overheads, and is thus the favored completion method in MPI libraries[4] and network benchmarks. Continuous polling consumes considerable host CPU resources, however.

Continuous network polling is also desirable for data services in some deployment scenarios, especially if service daemons can be isolated on dedicated nodes. It is less appropriate, however, if service daemons are colocated with other tasks or deployed in a power-constrained environment. Data services should gracefully idle when quiescent in these cases. As in the hot memory use case, this distinction can have a profound impact on network performance.

[4]Note that MPI implementations may elect to busy poll for progress even within blocking functions such as MPI_Recv() and MPI_Wait() without violating the MPI specification.

We use the `fi_msg_pingpong` benchmark from the libfabric fabtests to investigate this behavior. It measures round-trip latency at the libfabric API level and offers command line options to vary the completion method, as shown in Figure 4. In the default configuration, the benchmark busy polls for network completions. In the "fd" configuration, the benchmark blocks on file descriptors that are signaled by libfabric when network events are available. The latter method is also appealing for data services because it facilitates multiplexing events from multiple resources in a shared event loop (e.g., file and network I/O). This configuration more than triples the round-trip latency for libfabric, however, highlighting a case in which the default configuration of a microbenchmark does not necessarily measure the the implementation approach employed in an HPC data service. Annotations in the figure also show the median CPU utilization for each configuration.

C. CPU: core usage case study

Although network and storage devices are the most obvious factors in distributed storage service performance, the host CPU also plays a critical role because of its responsibility for coordinating devices and relaying data between them. Effective CPU usage is especially important for multithreaded data services executing on nodes with a large number of cores.

This case study highlights two specific issues that can obfuscate the impact of CPU performance in HPC data services. The first is that compute node resource managers often employ default policies that are optimized for application programming models rather that HPC data services. The second is that network and storage abstractions often do not clearly advertise their CPU requirements.

Figure 5 illustrates the confluence of both factors in a measurement of network performance on Theta. The baseline measurement shows point-to-point libfabric bandwidth, as measured by the `fi_msg_bw` benchmark from the fabtests collection, for a 32 KiB message size. The `fi_msg_bw`

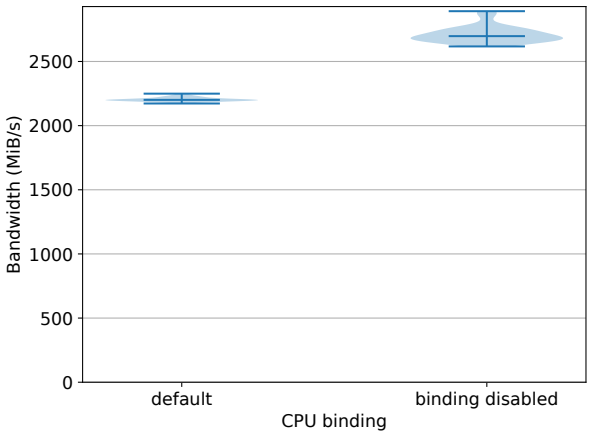

Fig. 5: Point-to-point internode libfabric bandwidth reported by `fi_msg_bw` with a 32 KiB message size on Theta. The "binding disabled" version executes the same benchmark with `--cc none` added to launcher command line.

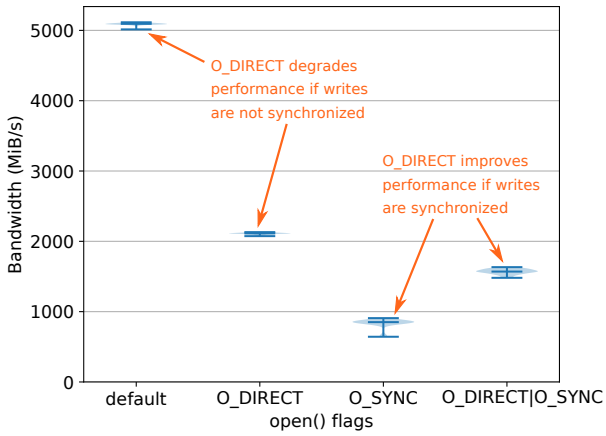

Fig. 6: Write bandwidth reported by `fio` with 16 threads and a 32 KiB access size on a Summit NVMe drive

benchmark is single threaded and would appear to place straightforward demands on the host CPU. However, the GNI libfabric provider spawns an internal progress thread by default; and as established in Section II-B, the fabtests benchmarks use CPU-intensive continuous polling by default. Taken together, this means that the benchmark needs two CPU cores to maximize bandwidth. Unfortunately, the `aprun` executable launcher binds processes to individual cores by default on this platform in a bid to improve pure MPI performance. When these factors are combined, the benchmark does not have sufficient CPU resources to maximize bandwidth. The "binding-disabled" configuration in Figure 5 adds `--cc none` command line argument to the aprun launcher to disable explicit core binding, thereby improving median benchmark performance by 22.5%.

D. Storage: caching case study

Caching is the most influential factor in storage device performance measurement. Cache settings must be chosen and documented with care when contrasting microbenchmark and service performance not only because of their performance impact but also because of their impact on durability, coherence, and fault tolerance. The most straightforward way to control file system cache behavior is by using optional flags to the `open()` system call. For example, the `O_DIRECT` flag can be used to bypass the kernel buffer cache[5]. The `O_SYNC` flag can be used to enforce that each individual write be made durable immediately. Other flags, as well as function calls such as `fdatasync()` and `fadvise()`, can also be used to control caching behavior.

These interplay of these cache settings can have subtle performance implications. For example, consider the scenario shown in Figure 6 where the experiment is intended to determine whether the O_DIRECT flag improves performance or not. Perceived write performance is highest with the default flags because the kernel buffer cache prevents the data from reaching the target storage device. This configuration measures system call and memory copy performance more so than storage device performance. The addition of the O_DIRECT flag bypasses the kernel buffer cache and forces each write to reach the device. However, it does not influence the storage device itself, which likely still employs an embedded write-back cache independent of the operating system. In contrast, the addition of the O_SYNC flag does not disable the kernel buffer cache, but it does immediately flush each write from both the kernel and device cache. This setting effectively turns the write-back storage cache into a write-through storage cache. O_SYNC and O_DIRECT can also be combined as in the last configuration to both disable the kernel cache and flush each write at the device level. Annotations in Figure 6 indicate how opposite conclusions are reached about the efficacy of O_DIRECT depending on the developer's cache settings.

E. Storage: file allocation case study

The O_DIRECT — O_SYNC results from Figure 6 in the previous section imply that a data service should be able to write 32 KiB entries to a durable log (e.g., for fault tolerance purposes) at a rate of 1.5 GiB/s. This interpretation overlooks secondary overheads, however, such as block allocation and size tracking.

We reconfigured `fio` to emulate a concurrent log update workload in order to investigate this phenomenon. In this case we use `libaio`[6] to minimize threading overhead while still performing 16-way concurrent writes (using the `fio` iodepth option) with O_DIRECT and O_SYNC. In this configuration, `fio` writes to a single shared file (e.g., a hypothetical data service log) rather than 16 separate files. The results are shown in Figure 7.

[5]O_DIRECT is a Linux-specific feature that is not defined by the POSIX specification. Its interpretation is therefore inconsistent across file systems, and it may impose additional alignment constraints.

[6]The Linux kernel on Summit does not support io-uring at this time.

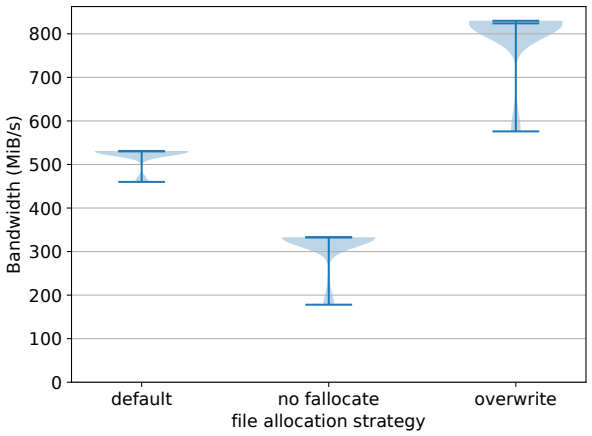

Fig. 7: Write bandwidth reported by `fio` with libaio, iodepth 16, and a 32 KiB access size on a Summit NVMe drive

The baseline measurement achieves a median of 530 MiB/s, which is only a third of that reported in the preceding section. This illustrates that concurrent updates to a shared file are not necessarily as efficient as concurrent updates to distinct files, even when taking precautions to minimize system call and buffer cache overhead.

The `strace` system call tracing tool also reveals unanticipated behavior from the `fio` benchmark itself: it calls `fallocate()` to preallocate the file prior to the timed write portion of the benchmark. This serves to isolate write performance from file allocation performance. In doing so, it likely produces a better representation of the pure write performance of the device, but it does not reflect the likely behavior of an HPC data service.

The file allocation step can be disabled via command line options; doing so produces the "no fallocate" measurements in Figure 7, which are an additional 200 MiB/s slower than the baseline. This is also the performance that a data service would achieve if it did not preallocate the log file. The discrepancy is caused by inefficiency in concurrent appends to shared files, which in turn is due to contention on block allocation and size updates. These are technically file system factors rather than storage device factors, but they impact practical storage service implementations nonetheless.

The impact of file allocation on write performance also hints at another potential benchmarking pitfall: is the file created from scratch during the benchmark or overwritten? The two preceding examples used a file size of 64 GiB and the "runtime" parameter to `fio` to measure sustained write performance for 10 seconds. This file size and time duration are sufficient to obtain a stable sustained write performance result. The third example, labeled "overwrite" in Figure 7, reduces file size to 1 GiB (e.g., to limit storage consumption by the benchmark): a configuration that would normally cause the benchmark to complete too quickly to produce stable measurements. However, this configuration has also been augmented with the `time_based=1` parameter, which instructs `fio` to iterate until the runtime has elapsed regardless of file size. This would appear to achieve the best of both worlds (sustained bandwidth measurement with low storage consumption), but the unintended consequence is that `fio` will continually wrap around at EOF and overwrite the file from the beginning without unlinking or otherwise reallocating the file. This configuration therefore not only preallocates the file as in the baseline measurement but in fact overwrites it repeatedly, thereby benefiting from additional caching effects on subsequent passes over the file and inflating the performance to a median of 824 MiB/s.

The violin plots in Figure 7 also reveal that shared file writes are more prone to bimodal performance than are the independent file writes that were measured in Section II-D. Most samples are near the peak (and median) performance, but some samples are significantly lower in each configuration.

III. Discussion and Conclusions

We investigated several common microbenchmarks in this study and uncovered implementation details that may surprise even experienced storage system developers. These surprises are a significant obstacle to successful performance engineering, especially when employing techniques such as roofline modeling. We believe roofline models offer great promise in improving the HPC storage community's understanding of storage system performance, but rigorous application of roofline modeling requires that we establish valid theoretical and empirical ceilings within the model. In attempting to apply roofline models to our own software, we realized that while existing microbenchmarks often do an excellent job approaching the best-case performance of the underlying hardware, the techniques used to achieve those performance levels are often inapplicable to the system usage requirements of storage system software.

Our study identified several potential pitfalls in the interpretation of widely used benchmarks for performance engineering in HPC data services. These pitfalls result from benchmark creators and service developers starting from different goals or different assumptions: Should operating system or user-space access be included in measurements? What configuration parameters are most relevant? What workload should be used?

No universal solution exists for establishing valid roofline model ceilings, but in order to lead the adoption of roofline models of performance analysis for data services we strongly encourage the HPC storage community to consider strategies to improve the existing state of the practice in I/O microbenchmarking. For example, microbenchmark authors could clearly document default benchmark settings (e.g., `fio` preallocation and OSU benchmark buffer reuse), HPC storage developers could clearly motivate the rationale for choosing particular benchmark configurations (e.g., why is a specific cache configuration relevant), and the community at large could pursue standardization of microbenchmarks for important motifs that reflect HPC storage service modalities. Initiatives such as these have the potential to greatly improve productivity in performance engineering for HPC data services.

ACKNOWLEDGMENTS

This material was based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. This manuscript has been approved for unlimited release and has been assigned LA-UR-20-26617.

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357, as well as resources of the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham, B. Robey, D. Robinson, B. Settlemyer, G. Shipman, S. Snyder, J. Soumagne, and Q. Zheng, "Mochi: Composing data services for high-performance computing environments," *Journal of Computer Science and Technology*, vol. 35, pp. 121–144, 2020.

[2] M. A. Sevilla, N. Watkins, I. Jimenez, P. Alvaro, S. Finkelstein, J. LeFevre, and C. Maltzahn, "Malacology: A programmable storage system," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 175–190. [Online]. Available: https://doi.org/10.1145/3064176.3064208

[3] Q. Zheng, C. D. Cranor, D. Guo, G. R. Ganger, G. Amvrosiadis, G. A. Gibson, B. W. Settlemyer, G. Grider, and F. Guo, "Scaling Embedded In-Situ Indexing with DeltaFS," in *Proceedings of the 2018 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 18)*, 2018, pp. 30–44.

[4] J. M. Wozniak, P. E. Davis, T. Shu, J. Ozik, N. Collier, M. Parashar, I. Foster, T. Brettin, and R. Stevens, "Scaling deep learning for cancer with advanced workflow storage integration," in *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, 2018, pp. 114–123.

[5] T. K. Petersen and J. Fragalla, "Optimizing performance of hpc storage systems," in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, 2013, pp. 1–6.

[6] M. Swan, "Tuning and analyzing Sonexion performance," in *Proceedings of the 2014 Cray User Group (CUG 2015)*, 2014, https://cug.org/proceedings/cug2014_proceedings/includes/files/pap119.pdf.

[7] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig, "Small-file access in parallel file systems," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–11.

[8] S. Williams, D. Patterson, L. Oliker, J. Shalf, and K. Yelick, "The roofline model: A pedagogical tool for program analysis and optimization," in *2008 IEEE Hot Chips 20 Symposium (HCS)*, 2008, pp. 1–71.

[9] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright, "A nine year study of file system and storage benchmarking," *ACM Transactions on Storage (TOS)*, vol. 4, no. 2, pp. 1–56, 2008.

[10] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems." ACM, 11 2015, pp. 73:1–73:12, proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15).

APPENDIX: ARTIFACT DESCRIPTION/ARTIFACT EVALUATION

SUMMARY OF THE EXPERIMENTS REPORTED

We executed a collection of component microbenchmarks (fio, osu_bw, fi_msg_pingpong, and fi_msg_bw) on the Summit supercomputer at the OLCF and the Theta supercomputer at the ALCF as described in the paper.

ARTIFACT AVAILABILITY

a) Software Artifact Availability:: There are no author-created software artifacts.

b) Hardware Artifact Availability:: There are no author-created hardware artifacts.

c) Data Artifact Availability: : All author-created data artifacts are maintained in a public repository under an OSI-approved license.

d) Proprietary Artifacts:: None of the associated artifacts, author-created or otherwise, are proprietary.

e) Author-Created or Modified Artifacts::

```
Persistent ID:
 ↪  https://doi.org/10.5281/zenodo.4000350
Artifact name: data artifact for "Keeping
 ↪  It Real: Why HPC Data Services Don't
 ↪  Achieve I/O Microbenchmark
 ↪  Performance"
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

f) Relevant hardware details:: We used two platforms for these experiments: the Summit supercomputer operated by the Oak Ridge Leadership Computing Facility and the Theta supercomputer operated by the Argonne Leadership Computing Facility. Summit consists of 4,608 compute nodes connected via EDR InfiniBand. Each Summit node contains 2 IBM Power9 CPUs, 6 NVidia Volta GPUs, 512 GiB of RAM, and a locally attached 1.6 TiB NVMe drive. Theta consists of 4,392 compute nodes connected by an Aries dragonfly network. Each node contains 64 Intel Knights Landing compute cores, 192 GiB of RAM, and a locally attached 128 GiB NVMe drive.

g) Compilers and versions:: XL 16.1.1-5, GCC 9.1.0, GCC 9.3.0

h) Applications and versions:: fabtests 1.10.1, osu-micro-benchmarks 5.6.3, fio 3.20

i) Libraries and versions:: libfabric 1.10.1, spectrum-mpi 10.3.1.2-20200121, cray-mpich 7.7.14

j) URL to output from scripts that gathers execution environment information:

```
https://anl.box.com/s/74z3ki88o4j2t756yra
 ↪  098rji02zm2mf
```