Extending I/O Observability to Explainability for Targeted Optimization

Neeraj Rajesh
Illinois Tech
nrajesh@hawk.illinoistech.edu

Sarah Neuwirth
Johannes Gutenberg University Mainz
neuwirth@uni-mainz.de

Jay Lofstead
Sandia National Laboratories
gflofst@sandia.gov

Xian-He Sun
Illinois Tech
sun@illinoistech.edu

Anthony Kougkas
Illinois Tech
akougkas@illinoistech.edu

Nikoli Dryden
Lawrence Livermore National Laboratory
dryden1@llnl.gov

Hariharan Devarajan
Lawrence Livermore National Laboratory
hariharandev1@llnl.gov

HPC I/O stacks are opaque. Existing tools such as Darshan count POSIX operations but stop at the syscall boundary, leaving thousands of ranks descending through VFS, page cache, and block layer with no end-to-end trace. The result is a wall of counters that records what happened but never explains why. A 12 ms `read()` is recorded in Darshan, yet the tool cannot say which layer stalled it. Without layer attribution, optimization chases symptoms rather than root causes, wasting weeks tuning the wrong parameters.

Fig. 1 illustrates the issue: POSIX-level tools see only the tip (`open`, `read`, `write`, `close`). Below the surface, every request traverses VFS, page cache, block layer, and NVMe. Each layer adds latency that POSIX counters cannot attribute. At scale, this opacity compounds: thousands of ranks with the same hidden descent produce aggregate counters that mask causation. A workload that appears I/O-bound at the POSIX layer may actually be page-cache-bound or block-queue-bound, but existing tools cannot tell which.

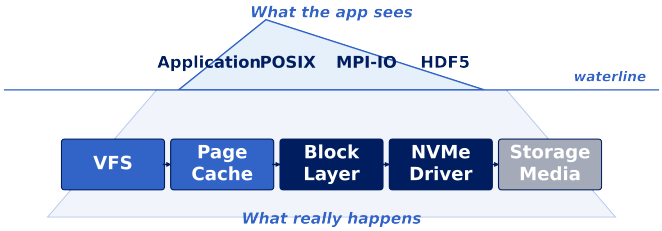


Fig. 1. The I/O Explainability Iceberg. POSIX-level tools see only the tip; below the surface, every request traverses VFS, page cache, block layer, and NVMe.

PEARL is an eBPF-based cross-layer tracing plugin for the DataCrumbs framework [1] that traces below the syscall boundary, extending [2]. It attaches tracepoints at every layer of the I/O stack (VFS, page cache, block layer, and NVMe) without modifying kernel source code or restarting services. A correlation engine stitches events by thread ID, timestamp, and matched pointers, fusing per-layer traces into one causal chain per request. An attribution map then assigns every microsecond to the layer responsible, producing a structured diagnosis report that enables root-cause analysis and informed tuning.

Darshan provides POSIX and MPI-IO counters with per-rank aggregates [3]. It records what happened but stops at the syscall, offering what Snyder et al. describe as a "limited perspective on I/O performance" because it cannot instrument the full stack end-to-end. PEARL provides VFS-to-cache-to-block-to-NVMe tracepoints with per-request causal timelines that trace end-to-end, from syscall to block I/O, to answer why a request was slow and where to fix it. Darshan gives aggregate latency; PEARL gives its layer-level cause. Configuration changes at the wrong layer are expensive and ineffective. Tuning block-scheduler parameters for a page-cache problem, or adjusting VFS cache pressure for an NVMe queue-depth issue, wastes time and resources.

To limit perturbation, PEARL uses selective activation: probes record detailed data only during anomalous periods, so tracing does not distort the workload it measures. The correlation engine produces compact causal chains rather than raw event dumps, keeping trace sizes manageable while preserving full attribution. Full eBPF tracing on 400K operations produces 530 MB traces; selective activation, common in I/O autotuning [4], yields compact output.

We evaluate PEARL on a Chameleon Cloud bare-metal (GigaIO) node running Ubuntu 26.04 with NVMe storage, using OpenMPI, Darshan, IOR, and PEARL. We test three IOR configurations: `fsync-heavy` (buffered I/O with frequent `fsync` calls), `large-direct` (large transfers using direct I/O, `O_DIRECT`), and `small-buf` (buffered I/O with small buffers and high operation count). First, Darshan reports end-to-end POSIX write latency of 0.45 ms to 5.88 ms, a total that spans user space, MPI, and kernel blocking. PEARL isolates the in-kernel CPU service time—the portion spent executing kernel code. The kernel itself consumes only 9.0 µs to 15.7 µs per request. The remainder is wait time in the block

layer and MPI synchronization; PEARL captures it in the causal timeline but attributes it separately from CPU time. In the buffered-I/O workloads, memory-management page faults (`handle_mm_fault`) triggered when accessing user-space buffers dominate that kernel time, accounting for 82% to 93%. This suggests that tuning page-cache pressure yields larger gains than block-scheduler adjustments for typical HPC checkpoint workloads, where buffered I/O dominates. In contrast, the `large-direct` workload shifts attribution to the block and NVMe layers, demonstrating that PEARL correctly distinguishes layer-specific bottlenecks across different I/O patterns. Second, PEARL overhead is negligible, with variance within system noise. Traces from 40K operations total only 130 MB.

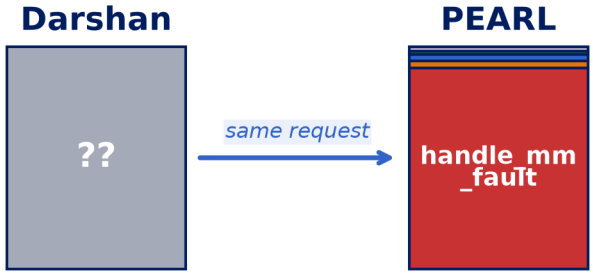


Fig. 2. Layer-level attribution from PEARL traces. The page-cache `handle_mm_fault` path dominates in-kernel CPU time for buffered-I/O workloads

Correlating events across layers is harder than collecting them. Matching a `bio` to `vfs_write` requires a thread-plus-timestamp-plus-pointer map that accounts for asynchronous I/O, request merging, and block-layer reordering, a challenge noted in bottleneck detection and tuning research [5]. This engineering complexity motivated PEARL's selective approach: trace only outliers, correlate into compact causal chains, and produce specific diagnosis reports rather than raw event dumps. Future work targets a closed loop: anomaly detection triggers detailed probes, causal explanation maps anomalies to layers, and targeted optimization converts diagnoses into configuration changes.

Kernel tracing moves HPC I/O analysis from opaque counters to causal explainability. By attributing latency to its source layer, PEARL enables optimization that targets the actual bottleneck rather than its symptoms, changing how we diagnose and fix storage performance problems in production environments.

References

[1] S. Neuwirth, H. Devarajan, C. Wang, and J. Lofstead, "XIO: Toward eXplainable I/O for HPC Systems," in *Proceedings of the 37th International Conference on Scalable Scientific Data Management*, 2025, pp. 1–6.

[2] H. Devarajan and K. Mohror, "Extracting and characterizing I/O behavior of HPC workloads," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, 2022, pp. 243–255.

[3] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular hpc i/o characterization with darshan," in *2016 5th workshop on extreme-scale programming tools (ESPT)*, 2016, pp. 9–17.

[4] B. Behzad, S. Byna, Prabhat, and M. Snir, "Optimizing i/o performance of hpc applications with autotuning," *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 4, pp. 1–27, 2019.

[5] J. L. Bez *et al.*, "I/o bottleneck detection and tuning: Connecting the dots using interactive log analysis," in *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*, 2021, pp. 15–22.