DataCrumbs: Efficient Cross-Layer Data Capture for Explainable I/O Performance in HPC Storage Stacks

Hariharan Devarajan*, Sarah Neuwirth†, Chen Wang‡, Ray A. O. Sinurat§, Neeraj Rajesh¶, Jay Lofstead‖, Vanessa Sochat*, Daniel Milroy*, and Thomas Scogland*

*Lawrence Livermore National Laboratory, Livermore, USA, {hariharandev1, sochat1, milroy1, scogland1}@llnl.gov

†Johannes Gutenberg University Mainz, Mainz, Germany, neuwirth@uni-mainz.de

‡Nanyang Technological University, Singapore, chen.wang@ntu.edu.sg

§University of Chicago, Chicago, USA, rayandrew@uchicago.edu

¶Illinois Institute of Technology, Chicago, USA, nrajesh@hawk.iit.edu

‖Sandia National Laboratories, Albuquerque, USA, gflofst@sandia.gov

Abstract*—Modern scientific workloads move large volumes of data through deep and complex I/O software stacks in high-performance computing (HPC) systems. Optimizing I/O performance requires visibility across the full I/O stack. State-of-the-art tools such as Darshan provide observability through performance counters, but they mostly capture API-level statistics that describe *what* happened rather than *why* it happened. As a result, performance variation in otherwise similar I/O operations is often dismissed as system jitter without direct evidence. Emerging approaches such as eBPF make it possible to collect fine-grained data with low overhead, but naively doing so produces an overwhelming volume of information. To address this problem, we present DataCrumbs, an I/O monitoring tool for explainable I/O analysis in HPC systems. DataCrumbs leverages function probing to collect selective fine-grained performance data across the full I/O stack (from user-space runtimes and libraries to kernel-space file-system paths) to reveal *why* performance varies. First, it flexibly attaches to multiple I/O software components, instrumenting 8,723 functions with only 78 ms startup overhead. Second, it captures both user- and kernel-space functions with just 0.1% profiling overhead and under 3% tracing overhead (1.8% at 1M events), compared with 3% for Darshan profiling and ∼9% for Darshan DXT tracing. Third, a time-windowed, embarrassingly parallel event-association algorithm reduces cross-layer attribution cost from $\mathcal{O}(n^2)$ to $\mathcal{O}((n/t)^2)$, enabling scalable root-cause analysis across I/O stack layers. Fourth, across four use cases (IOR, DLIO, VPIC+BD-CATS, and Montage), DataCrumbs exposes actionable, cross-layer insights not accessible to user-space-only tools; acting on one such insight, we tune OrangeFS read-ahead and prefetch hints for Montage, reducing its POSIX I/O time by 1.35×. Overall, DataCrumbs enables selective, fine-grained monitoring for explainable I/O analysis across real HPC workloads.

***Index Terms*—DataCrumbs, eBPF, kprobes, I/O performance, performance explainability, Multi-layer I/O monitoring, I/O stack**

I. Introduction

HPC systems rely on two main classes of I/O monitoring tools: *profilers* and *tracers* [1], [2]. Profilers [3] collect coarse-grained statistics, such as I/O counts, transfer sizes, and per-file access patterns, at overhead as low as 3% [3], suiting facility-wide deployment [4]–[6]. Tracers [7]–[9] instead record individual I/O events for finer-grained bottleneck analysis, at higher overhead of 21% [8]–26% [10]. Data from these tools underpins analysis frameworks [11], [12] and empirical studies [13]–[16] characterizing problematic I/O behavior in production workloads.

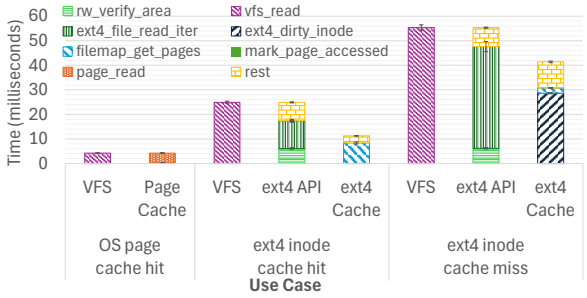


Fig. 1. Fine-grained kernel-stack monitoring shows a 14× latency difference for similar read requests served from different layers: 2 ms from the OS page cache, 24 ms from the ext4 inode cache, and 55 ms from the device.

Modern data-intensive scientific workloads need visibility across the full I/O path: application and runtime, high-level interfaces and middleware (e.g., HDF5, MPI-IO), user-space POSIX/libc calls, and kernel-space file-system, cache, and block-I/O paths. This matters because I/O often exceeds 15% of application runtime [7], [17]–[19], and effective optimization depends on knowing where that time goes across layers. Most existing tools instrument only user-space *libc* APIs, revealing *what* a workload does (e.g., I/O time, operation counts, bandwidth) but rarely *why* a behavior occurs. Deeper instrumentation via LD-style wrapping and runtime interception grows increasingly expensive, so root-cause analysis is left to I/O experts who must formulate and test hypotheses manually, a process made harder by HPC systems' restricted user-level access.

For example, two identical read requests to the same device can show substantially different latency, often attributed to system jitter, though the true cause may instead be page-cache effects [20], ext4 inode-cache effects [21], or genuine jitter [22]. As Figure 1 shows, a 64 KB read may complete in 2 ms from the page cache, 24 ms through the inode cache, or 55 ms from the device, a 14× difference, requiring visibility into counters and execution paths that user-space tools do not expose. More broadly, API-level monitoring cannot tell when lower layers

SC26 Workshops, November 15-20, 2026, Chicago, Illinois, USA
979-8-3195-1221-5/26/$31.00 ©2026 IEEE

are bypassed, when kernel activity dominates end-to-end cost, or when internal mechanisms like MPI-IO's two-phase I/O shape the observed behavior [23], [24], so identifying root causes requires monitoring the deep I/O stack: runtimes, I/O interfaces, middleware, and kernel-space functions [25].

Designing such a monitoring tool raises three challenges: (1) the relevant software components and routines must be discovered across a heterogeneous and evolving I/O stack; (2) observations from multiple layers must be integrated so that causal relationships can be established rather than inferred from isolated counters; and (3) capturing thousands of functions can introduce substantial runtime overhead if collection and storage are not carefully designed.

Although eBPF (extended Berkeley Packet Filter) enables low-overhead instrumentation, naively collecting everything it can observe quickly produces MB/s of trace data, much of which is irrelevant for explaining performance. Explainable monitoring therefore depends not only on visibility, but also on selective data collection. We present DataCrumbs, an I/O monitoring tool for explainable I/O performance analysis in HPC storage stacks. By explainable, we mean that DataCrumbs performs cross-layer performance attribution (Algorithm 1): it links a target API event to the overlapping lower-layer events that account for its cost, rather than merely reporting that a slowdown occurred; this cross-layer attribution is the specific mechanism used throughout this paper, while more general data-driven attribution (e.g., structural causal modeling) beyond directly observed links remains future work. DataCrumbs dynamically discovers software routines using language-specific methods, such as ELF analysis for C/C++ and USDT hooks for Python, and inserts lightweight probes to collect fine-grained data efficiently. To minimize interference, it buffers collected events in kernel space and transfers them asynchronously to user space, thereby decoupling data capture from storage. By correlating performance data across user and kernel layers, DataCrumbs extends the observability available to analysis frameworks such as [11], [13] by providing kernel-level data that user-space-only tools cannot expose. DataCrumbs requires elevated privileges for eBPF probe attachment, which is handled transparently through scheduler prolog and epilog scripts (Flux and Slurm) that automatically deploy it on user request from a probe list pre-vetted by system administrators. This deployment model is already approved and running on two HPC testbeds. The main contributions of this work are:

1) We design an *automated function-discovery mechanism* that identifies and instruments I/O routines across heterogeneous HPC software stacks using ELF symbol analysis, environment module detection, Spack integration, and language-specific hooks.
2) We develop a *low-overhead probing mechanism* that decouples performance-data capture from storage using assembly-level trampolines and asynchronous kernel-to-user-space data transfer.
3) We introduce a *time-windowed, embarrassingly parallel event-association algorithm* that reduces cross-layer attribution from $\mathcal{O}(n^2)$ to $\mathcal{O}((n/t)^2)$ by partitioning events into independent time windows, enabling scalable root-cause analysis across I/O stack layers.
4) We demonstrate *four representative use cases* from HPC benchmarks (IOR, DLIO) and scientific workflows (VPIC+BD-CATS, Montage) that expose actionable, cross-layer insights not accessible to user-space-only tools.

II. Related Work

We review prior monitoring tools for HPC I/O analysis and highlight how DataCrumbs differs.

A. Traditional HPC Monitoring Tools

Traditional monitoring tools emphasize coarse system metrics such as bandwidth, IOPs, and file-level counters. At the system level, Lustre Monitoring Tool [26] (LMT), mmpmon [27], and FSMonitor [28] provide scalable monitoring for specific file systems, effective for health monitoring and bottleneck detection but reporting system-level counters rather than causally linking behavior across application, library, and kernel layers. DataCrumbs, in contrast, targets explainability by correlating events across those layers.

At the application level, Darshan [29], Score-P [30], Recorder [31], and DFTracer [7] provide profiling and tracing for HPC applications: Darshan offers low-overhead summaries at interfaces such as HDF5, MPI, POSIX, and STDIO; Score-P and Recorder collect richer traces; and DFTracer extends tracing to AI workflows and I/O middleware. However, all are fundamentally limited to user-space collection, so they cannot observe kernel-space behavior or fully attribute costs across user-space libraries, middleware, and file-system paths. DataCrumbs extends this visibility into kernel space while preserving low overhead.

B. Emerging eBPF-based Monitoring Tools

Recent eBPF-based tools move closer to deep, low-overhead observability. EZIOTracer [10] unifies kernel- and user-space I/O tracing at 15% (profiling) to 26% (tracing) overhead on buffered I/O workloads with 32 threads, but its use of BCC's `perf_buffer` with synchronous polling prevents decoupling capture from processing, causing buffer saturation with event loss and high overhead under buffered I/O. DataCrumbs, in contrast, uses libbpf's `ring_buffer` with async kernel-to-user transfer, safely decoupling collection from storage. DataCrumbs achieves 0.1% profiling overhead versus 15% for EZIOTracer (a 150× improvement); for tracing, DataCrumbs incurs under 3% overhead versus 26% for EZIOTracer and ∼9% for Darshan DXT (Table II and Table III, respectively).

Tetragon [32] provides eBPF-based security observability for cloud systems, not HPC I/O; Process Monitor for Linux [33] traces system calls but isn't designed for cross-layer performance explanation; and the XIO prototype [34] captured C++ and kernel data but relied on single-producer queues. DataCrumbs instead provides deployable infrastructure for explainable I/O with dynamic function discovery, multi-language support (C/C++, Fortran, Python), and efficient multi-producer ring buffers.

III. Background

We briefly summarize the eBPF functionality that enables low-overhead metrics collection and why the HPC I/O stack is a demanding target for it.

A. Extended Berkeley Packet Filter (eBPF)

Extended Berkeley Packet Filter (eBPF) [35], [36] runs sandboxed programs in kernel space without modifying kernel code or loading modules, and is widely used for networking, security, and low-overhead observability [32], [37]–[40]. In HPC, it enables fine-grained monitoring of system calls, network events, and hardware activity for performance tuning, with JIT compilation keeping collection efficient [35]. Developers typically build on tools such as BPF Compiler Collection (BCC) [41], BPFTrace [42], BPFTime [43], and BPF CO-RE [44], trading off control, portability, and usability.

Our work, DataCrumbs, builds on this ecosystem for explainable I/O. Rather than using a static configuration, it evaluates each application environment and deploys targeted eBPF hooks across libraries, middleware, and the kernel so bottlenecks can be diagnosed across the full stack.

B. I/O Stack Obscures Explainability

The HPC I/O stack is deeply layered, spanning interfaces such as POSIX and STDIO (libc), higher-level libraries such as HDF5 [45], PnetCDF [46], and ADIOS [47], middleware such as UnifyFS [48] and Hermes [49], and kernel and distributed file systems such as ext4, XFS, GPFS [50], and Lustre [51]. These layers span multiple nodes and often multiple networks, making I/O behavior difficult to explain from a single vantage point. While each DataCrumbs instance collects data per node independently, correlating the per-node logs reveals cross-node path information. Cross-node event correlation is currently performed offline after job completion; online cross-node correlation is left for future work.

I/O tuning is difficult because these layers are interdependent [52], [53]. For example, PnetCDF depends on MPI-IO, which depends on POSIX and the underlying file system, so metadata layout, file locking, consistency semantics, aggregation strategy, and striping decisions can all change end-to-end behavior [23], [54]. Internal library complexity adds another source of opacity: HDF5 metadata management, chunking, and collective I/O all affect storage behavior [53], while MPI-IO's ROMIO layer exposes tunable aggregation choices with nontrivial trade-offs [24]. As a result, isolated single-layer monitoring cannot explain I/O performance. Explainability requires comprehensive cross-layer data, often collected across many nodes and networks.

IV. Design

We introduce DataCrumbs [55], an I/O monitoring tool designed to collect detailed performance data from the entire I/O stack and thereby enabling a deeper understanding of observed I/O behavior in workloads. A central design principle of DataCrumbs is to capture performance data across all layers of the I/O software stack, from applications and runtimes to middleware, file systems, and kernel I/O paths, so that behavior observed at one layer can be causally linked to activity at another. This multi-layer visibility is essential for explainable I/O because performance variability often originates outside the layer where it is observed. Our approach is guided by three core objectives:

1) supporting diverse software layers by dynamically extracting functions across different toolchains,
2) maintaining low overhead to minimize impact on workload performance,
3) optimizing collection for efficient large-scale trace analysis.

These objectives directly shape the design of DataCrumbs: automated software discovery supports broad coverage, decoupled collection reduces runtime overhead, and structured multi-layer traces enable scalable explainable I/O analysis.

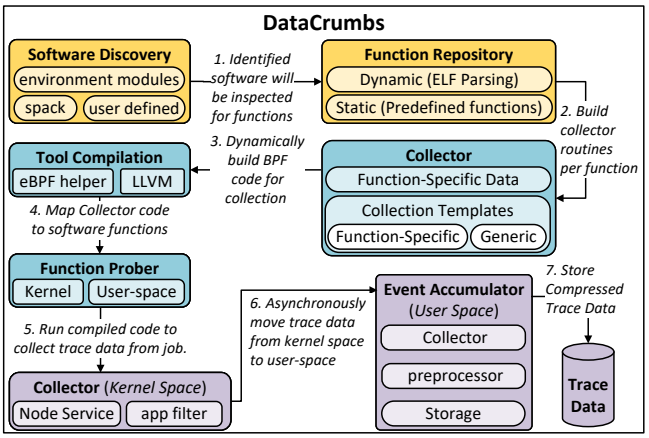


Fig. 2. DataCrumbs's high-level design for flexible, efficient, and comprehensive performance data collection in HPC storage software stacks.

A. High-level Overview

DataCrumbs consists of seven core components that capture, process, and store performance data in HPC workloads (Figure 2). Upon job allocation, it initializes across all nodes, detecting loaded software through environment modules and Spack [56] to populate a customizable collection catalog. Once identified, DataCrumbs extracts functions from the software (*Step 1*) via strategies such as Executable and Linkable Format (ELF) parsing and predefined Python hooks, then builds function-specific collector routines (*Step 2*) that capture function arguments for tracing or high-level statistics for profiling; predefined templates minimize the number of probed functions, keying DataCrumbs's ability to collect only the data needed for later analysis. Collector functions are compiled into kernel code using LLVM (*Step 3*) via the BPF Compile Once - Run Everywhere (BPFCORE) toolkit, leveraging predefined helpers to capture performance data at function entry and exit. DataCrumbs then attaches probes at function boundaries (*Step 4*) using `k-probes` for kernel space and `uprobes` for user space; because this requires elevated privileges, we integrate DataCrumbs with the scheduler's prolog/epilog so users need only request profiling for the system to deploy it automatically on each node.

Once deployed, DataCrumbs runs as a kernel-space service (*Step 5*) that captures data from all processes on the node; a lightweight library provides two custom functions to identify processes of interest, so only relevant application or system software is profiled. During execution, DataCrumbs stores performance data in kernel space, using maps for profiling mode and an event ring buffer for tracing mode, while a user-space thread asynchronously reads, processes, and writes this data to the filesystem (*Step 6*). When the workload completes, DataCrumbs waits a configurable period before compressing the trace and terminating (*Step 7*), completing an efficient, low-overhead workflow that adapts to different workloads.

Together, these components enable DataCrumbs to collect causally linked performance data across the full I/O stack. We describe this capability in more detail in Section IV-D, where we present the multi-layer view and trace organization that support explainable I/O analysis.

B. Hooking into Deep I/O Software Stacks

Understanding software behavior is essential for optimizing HPC performance and diagnosing inefficiencies, but manually identifying relevant components and extracting function-level details is complex and time-consuming given the diversity of HPC software stacks and programming models. DataCrumbs addresses this with an automated approach for discovering software and extracting function information.

1) Identifying Software: First, DataCrumbs leverages environment variables to identify loaded software modules, running `module list` for active modules and `module whatis` for their metadata, extracting the library path variable that points to shared object (`.so`) files and locating them with `find`. Second, it integrates with Spack by executing `spack find` to list installed packages and inspecting each package's library folder for associated `.so` files, ensuring compatibility with Spack-managed HPC stacks. Finally, users can explicitly specify object files so software missed by modules or Spack can still be included. Combining these three methods, DataCrumbs automatically extracts function details using predefined system calls, ELF symbol analysis for C/C++ binaries, and runtime hooks for Python, CUDA, and HIP. Interfaces are not hard-coded: any discovered library (e.g., ADIOS2, PnetCDF) is instrumented the same way. We evaluate HDF5 and MPI-IO because our workloads use them.

2) Identifying Functions: First, DataCrumbs includes predefined functions for system-level I/O: the Virtual File System (VFS), OS caching, file systems such as XFS, ZFS, and ext4, and the Block I/O layer. Second, it tracks standard libc I/O functions such as POSIX and STDIO APIs, frequently used by HPC applications, capturing critical I/O behavior without additional discovery steps. Third, for C/C++ executables and shared libraries, DataCrumbs extracts function names from ELF symbols, retrieving precise signatures and locations when debugging symbols are present. Finally, for Python, it uses interpreter entry and exit hooks to capture workload-specific functions, enabled at Python build time. DataCrumbs extends this same binary-level approach to Fortran applications via `uprobes`, which trace subroutine and function entry/exit directly on the compiled binary without requiring language-specific instrumentation. Combining these approaches, DataCrumbs automatically discovers I/O software and tracks function execution across diverse stacks, keeping log sizes manageable while capturing all the data needed to understand *why* I/O behaves as observed.

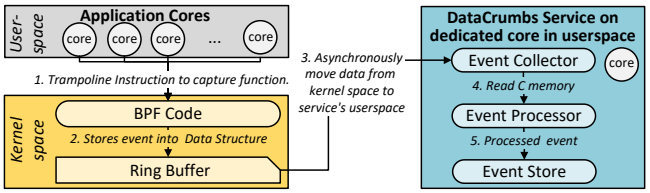


Fig. 3. High-level overview of DataCrumbs's decoupled runtime, enabling function tracing with low overhead.

C. Efficient Decoupled Runtime Collection

Collecting performance data at runtime is essential for understanding HPC system behavior, but traditional tracing introduces significant overhead as collected-event volume grows. DataCrumbs addresses this with a decoupled collection mechanism: instead of synchronously processing each event, which scales poorly, it separates capture from processing and storage, immediately storing relevant function calls in lightweight kernel-space in-memory buffers and processing them asynchronously, independent of storage. This reduces tracing overhead to under 3% (versus 26% for EZIOTracer) while keeping system performance stable and responsive as traced events grow. Isolating each node also reveals performance variations across the application workflow, further explaining why performance differs; correlating operations between nodes needs time and sequencing information from node-specific logs, which we perform offline to avoid runtime impact.

Figure 3 illustrates this decoupled collection workflow. DataCrumbs attaches jump instructions (trampolines) at the assembly level to capture function call boundaries (*Step 1*); this low-level probing is inherently language-agnostic, since all languages compile down to assembly. Captured events are stored in efficient kernel-level Maps and Ring Buffers (*Step 2*), structures designed for low-cost insertions and lookups suited to high-frequency, temporary storage. Rather than processing events synchronously, a user-space engine asynchronously reads data from these structures (*Step 3*), so processing and storage overhead does not directly affect application execution; the processing rate must exceed the collection rate to avoid buffer overflow, and DataCrumbs keeps processing residual kernel-space data even if the application terminates, though a node failure loses unflushed data. Transferred data is converted from raw bytes to structured C types (*Step 4*), then resolved and optimized for efficient storage (*Step 5*) before being written to a per-node trace file. Separating capture from processing and storage keeps runtime overhead low while enabling scalable HPC performance monitoring.

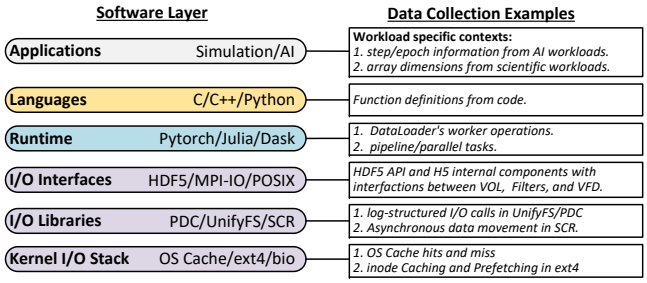


Fig. 4. DataCrumbs capturing multi-layer performance data across HPC storage systems

D. Performance Data Collection

We target three aspects of comprehensive performance data collection in HPC storage systems: capturing data across multiple layers (system I/O calls, file systems, and the Block I/O layer); two collection modes, *profiler* for low-overhead aggregated statistics and *tracer* for fine-grained event details; and an efficient trace format optimized for compact storage and fast retrieval. Together, these support scalable, efficient monitoring while preserving the detail needed for explainable I/O analysis.

1) Capturing performance data from multiple layers: DataCrumbs enables comprehensive performance data collection across multiple layers of the HPC storage software stack (Figure 4), which shows a typical stack spanning application layers, programming languages, runtimes, I/O interfaces, middleware libraries, and the kernel. The key challenge is determining which information is most relevant for understanding I/O performance, since relevant signals vary by layer and software context.

Each layer offers unique insight. At the application level, we capture function execution and workload-specific context, such as scientific data dimensions, AI training steps, or simulation iterations, critical for diagnosing bottlenecks. At the language level, we extract function signatures, parameters, dependencies, and coverage to analyze how behavior propagates across layers. The runtime level adds execution-specific detail, e.g., data-loading efficiency in PyTorch, or task pipelining and parallelism metrics in task-based systems like Julia and Dask. At the I/O interface level, we track interactions among components, such as HDF5 calling MPI-IO, which in turn invokes POSIX, revealing how optimizations like chunking, filters, and data-type conversions affect lower layers and overall storage performance. Monitoring function calls within I/O libraries such as UnifyFS [48], SCR [57], and PDC [58] further shows how effectively these libraries optimize application I/O and whether they behave as expected. Finally, at the kernel I/O stack, we capture OS cache hit/miss rates, ext4 prefetching, and block I/O performance for detailed system-level assessment.

By analyzing each layer both individually and in relation to the others, we obtain fine-grained system insight together with high-level workload context. This layered view supports not only observability, but also the causal links required for explainability of I/O system behavior.

2) Profiler and Tracer Mode: Different types of performance analysis require different levels of data granularity. Some use cases, including understanding the overall contribution of a function to system performance, benefit from aggregated high-level statistics, while others, such as modeling function hierarchies and dependencies, require fine-grained tracing. To support both scenarios, DataCrumbs offers two distinct modes: profiler and tracer, allowing users to choose the appropriate collection granularity for their analysis needs.

In profiler mode, DataCrumbs collects event data such as elapsed time and function call frequency, storing this information in per-time-interval and per-process maps. This mode is efficient and incurs at most 0.1% overhead on application runtime (refer to Table II), making it well suited for monitoring system-wide performance with minimal interference. In contrast, tracer mode records detailed event-level information and function call sequences in a ring buffer, enabling precise modeling of function execution but incurring higher overhead (under 3% as shown in Table III). Overheads are measured by comparing workload runtimes with and without DataCrumbs enabled.

Choosing between these modes depends on the analysis objective. Tracing is essential for exploratory analysis, where identifying critical functions and understanding their precise impact is necessary. Profiling, in contrast, provides a broader view of workload behavior and is therefore well suited for continuous system monitoring. Given this flexibility, DataCrumbs can also be deployed as a system-wide profiler with selective tracing enabled for specific workloads.

3) Trace format to handle performance data: Capturing comprehensive performance data in HPC storage systems generates an enormous trace volume, potentially 1000× more than tools such as Darshan, so efficiently storing and analyzing it requires a format that minimizes overhead while supporting fast retrieval and scalable analysis. Our trace format is designed around three objectives:

1) *Buffered Writing:* Because trace data is written at high frequency, we use buffered writing to flush data to persistent storage in batches and improve write performance.
2) *No Event Dependencies:* Each event is stored independently, without ordering or dependency constraints, allowing embarrassingly parallel processing.
3) *Analysis-Friendly Structure:* The trace format is optimized for parallel parsing, enabling scalable analysis using tools like Dask [59]. This ensures that large-scale performance data can be processed efficiently without bottlenecks.

Unlike OTF2 [60], whose streams are decoded sequentially per location, each DataCrumbs event is self-contained, so indexed chunks load in parallel with Dask and open directly in Perfetto. DataCrumbs also retains DFTracer's gzip compression with indexing, enabling fast querying and parallelized trace loading, keeping the format efficient and scalable for high-performance, comprehensive analysis.

Algorithm 1 Time-windowed event association for explainable I/O

Require: Events E sorted by $(pid, thread, t_s)$, target API events A, window expansion factor Δ

Ensure: Associated event groups G and per-group contribution tables C

Initialize empty groups $G \leftarrow \emptyset$
for all $a \in A$ **do**
  Extract process/thread scope $(pid_a, thread_a)$ and interval $[t_s(a), t_e(a)]$
  Expand interval to $I_a = [t_s(a) - \Delta,\ t_e(a) + \Delta]$
  Query candidate events $E_a = \{e \in E \mid pid(e) = pid_a \wedge thread(e) = thread_a \wedge [t_s(e), t_e(e)] \cap I_a \neq \emptyset\}$
  Assign a unique group id g_a and set $G[g_a] \leftarrow E_a$
end for
Execute each $G[g]$ in parallel workers
for all group g in parallel **do**
  Build parent-child relations by interval containment/overlap within $G[g]$
  Topologically order events to derive serialized critical paths
  Compute per-event contribution scores $C[g][e]$ from exclusive and inclusive time
end for
return G, C

E. Performing Explainable I/O Analysis

To connect relevant events across the stack, we analyze DataCrumbs traces in Dask DataFrame [59], first filtering key API-layer events, then retrieving associated events in the same execution context. Since a naive overlap search is expensive ($O(n^2)$ for n events), we instead use the time-windowed parallel event-association algorithm in Algorithm 1 to group related events efficiently while preserving accuracy, making root-cause analysis feasible at large scale.

Figure 5 illustrates the analysis outcome, and Algorithm 1 defines event grouping and contribution computation. The method assumes short temporal correlation between cause and effect. When asynchronous behavior introduces longer delays, we increase the expansion factor Δ to preserve correct associations. The workflow is: mark API calls that require explanation (Step 1), collect overlapping events within each call's process-local time window and assign group IDs (Step 2), derive event hierarchy/serialization inside each group (Step 3), and compute per-event contribution to the target API call (Step 4). These four steps correspond directly to the ext4 trace shown below the algorithm in Figure 5: Steps 1–2 produce the grouped, overlapping events plotted along the timeline, while Steps 3–4 produce the hierarchical parent/child contribution breakdown shown beneath it. This reduces effective complexity to $O((n/t)^2)$, where t is the number of time windows, and enables embarrassingly parallel execution across distributed Dask workers.

This analysis links DataCrumbs data across user libraries, middleware, file systems, and kernel layers to identify bottlenecks and their causes, exposing cross-layer interactions that

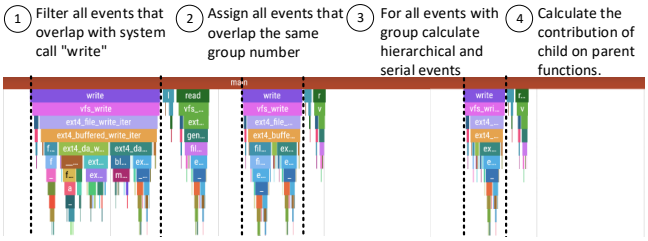


Fig. 5. The associative algorithm identifies target API's related events, shown on a DataCrumbs ext4 trace in Perfetto, enabling explainable I/O analysis.

isolated views miss and improving explainability for optimization decisions, while the time-windowed parallel algorithm keeps the method scalable for large traces. Exploring all tool combinations remains future work; here we focus on collecting the data and organizing it for scalable downstream analysis.

F. Scalability Considerations

A critical aspect of DataCrumbs is that it operates as a fully decentralized, per-node monitoring service: unlike distributed monitoring systems requiring cross-node coordination, each node independently captures, processes, and stores its own performance data. Scaling a workload from tens to thousands of nodes therefore does not increase monitoring overhead on any individual node, suiting both small clusters and extreme-scale systems.

Pre-deployment Configuration. DataCrumbs deploys as a per-node service through scheduler prolog/epilog scripts, or as a systemd service for a job allocation's lifetime; each compute node starts its own instance at job start and terminates it at job end. Performance data is written to per-node log files on shared storage such as NFS or Lustre, but the service itself needs no cross-node coordination, keeping deployment simple and avoiding additional synchronization overhead.

eBPF Collection Scalability. Data collection is performed locally in each node's kernel using `k-probes` and `uprobes`. The probes capture function events and write them into low-latency kernel Maps and Ring Buffers, so insertion cost is paid locally without any cross-node communication. Because eBPF executes entirely within the kernel on each node, the collection overhead remains predictable as system size grows; adding more nodes replicates the same local monitoring cost rather than increasing it. Within a node, eBPF observes every process through the single per-node DataCrumbs service rather than one instance per process, and the underlying ring buffer is a multi-producer, multi-consumer structure, so it naturally scales to concurrent processes; the overhead numbers reported in Section V-A already reflect all cores of a node driving I/O concurrently through this one service.

Post-Collection Processing Scalability. Captured data is asynchronously transferred to a user-space service that formats and stores traces without blocking the application. Running independently on each node, this service reads local kernel buffers, converts raw data into structured events, and writes per-node traces; configurable thread pools, buffer sizes, and processing rates let it keep pace with collection while preserving the capture/storage decoupling. This separation

matters for scalability because processing cost grows with local event volume, not with the job's total node count.

Post-Processing Analysis Scalability. After job completion, DataCrumbs produces independent per-node traces analyzable offline with frameworks such as Dask. Since the trace format is optimized for parallel parsing with no cross-node dependencies, post-processing can be distributed across many workers, even on separate analysis infrastructure, decoupling analysis scalability from the monitored job's size and letting large runs be processed efficiently through embarrassingly parallel execution.

G. Use Cases and Practical Optimizations

DataCrumbs enables collection of data that pinpoints root causes down to specific software layers and function calls, which in turn supports diagnostic tools that recommend targeted, data-driven optimizations. Because profiling is performed over time intervals and the ring-buffer design decouples collection from processing, these insights can be derived in real time without adding overhead to the application itself. The practical decisions enabled by these insights are:

1) ***Application/Middleware-level:*** automatically tuning and configuring HDF5 (including MPI collective metadata updates) based on the application's measured performance.
2) ***Client node-level:*** tuning caches across the storage software stack by adjusting parameters such as the OS page cache percentage, the number of client pages, and the size of the HDF5 chunk cache, while also quantifying cross-layer cache effects by attributing performance for the stack.
3) ***System-level:*** tuning parallel file system settings (e.g., Lustre client RPC limits, server redundancy, and compression) using measured bottlenecks rather than assumptions.

These optimizations are driven by measured performance data rather than assumptions about system behavior.

V. Evaluation

This section validates the three technical contributions from the Introduction: (1) automated function discovery across heterogeneous I/O stacks, (2) low-overhead decoupled collection, and (3) cross-layer event association for root-cause analysis; Contribution 4 (representative use cases) is demonstrated through the benchmark and workflow case studies below. We evaluate flexibility (probe attachment cost), overhead (profiling and tracing), and explanatory coverage (benchmark and workflow use cases).

A. Methodology

This subsection summarizes the testbed, software, workloads, and performance data used in the evaluation.

1) Testbed: All experiments were conducted on the Ares Research Cluster testbed [61]. We use 32 compute nodes for application runs; each node has an Intel Xeon Silver 4114 @ 2.20 GHz, 96 GB DDR4 memory, and a 512 GB NVMe SSD. The OrangeFS storage pool is built from 16 storage nodes, each with dual AMD Opteron 2384 @ 2.7 GHz processors, 32 GB DDR3 memory, and a 2 TB HDD.

2) Software: DataCrumbs uses BPF CO-RE [44] (v0.30.0) built on eBPF (Kernel 5.6) and requires CMake 3.4 for compilation and Python 3.10 for execution. The setup also includes GCC 11.4.0, OpenMPI 5.0.5, and HDF5 1.14.5. For trace analysis, we use Dask and Perfetto UI for visualization (e.g., as shown in Figure 5). For a fair comparison, baseline tools were configured to match DataCrumbs's buffering: Darshan and Score-P use the same kernel-side ring-buffer size, and EZIOTracer uses the same per-CPU buffer as DataCrumbs. This makes the comparison conservative for DataCrumbs, which collects over 1000× more events than any of the compared tools.

3) Workloads: We evaluate with two benchmarks (IOR and DLIO) and two real workflows (VPIC+BD-CATS and Montage). IOR [62] uses HDF5 on a single node with 40 processes, a file-per-process pattern, and sequential writes of 1 GB blocks with 1 MB transfers to generate sustained I/O events. DLIO runs the Unet3D workload on 32 nodes with 4 GPU processes and 44 I/O workers per node, staging the dataset into local SSDs before training. VPIC+BD-CATS is a producer-consumer workflow on 32 nodes with 40 processes per node, while Montage is a multi-stage FITS-to-PNG workflow executed on up to 32 nodes with data-parallel stages. For IOR and DLIO, each test runs ten times and figures report mean and standard deviation. We compare DataCrumbs primarily against Darshan; Score-P and Recorder expose similar information with higher overhead.

4) Performance Data: DataCrumbs operates in the profiler and tracer modes described in Section IV-D.

B. Handling the Deep I/O Software Stack

Using the IOR configuration from Section V-A3, we evaluate the runtime cost of attaching probes at different software layers, from selective I/O calls to MPI and HDF5. DataCrumbs was configured to profile each layer using functions discovered from the libraries' ELF symbols.

The evaluation uses two metrics: runtime to attach function probes and the number of functions profiled, across four cumulative configurations (Table I). *Selective I/O Calls* (100 chosen system calls and kernel functions) require only 448 µs to attach probes. Expanding to *All POSIX + Kernel* (2,489 functions) raises overhead to 24 ms, showing that broader coverage adds cost; adding MPI-IO (3,544 total) raises it to 33 ms, while the most extensive configuration, HDF5 (8,723 functions), incurs the highest overhead at 78.9 ms.

Although overhead grows with more instrumented functions, the growth is nonlinear, with diminishing per-function cost at higher layers: profiling additional software layers adds overhead, but the relative cost decreases as the function set grows. For HPC workloads that typically run for hours, this millisecond-scale probe-attachment overhead is negligible relative to overall job execution time. Manually identifying and instrumenting these functions is also infeasible given the number of routines across the stack, highlighting a key advantage of DataCrumbs: it automates function discovery and instrumentation, making fine-grained

TABLE I

DATACRUMBS'S PROBE-ATTACHMENT OVERHEAD GROWS WITH THE NUMBER OF INSTRUMENTED FUNCTIONS, BUT THE GROWTH IS SUBLINEAR, WITH DIMINISHING PER-FUNCTION COST.

	Selective I/O Calls	All POSIX + Kernel	MPI	HDF5
Runtime (ms)	0.448	24.321	33.041	78.980
Number of APIs	100	2,489	3,544	8,723

TABLE II

PROFILING OVERHEAD: DATACRUMBS CAPTURES >1000 MORE EVENTS WITH LOWER OVERHEAD THAN DARSHAN, SCORE-P, AND EZIOTRACER, WITH TYPICAL VARIABILITY < 1%.

Tool	10K Events / s	100K Events / s	1M Events / s
Base	0.252±0.001	2.419±0.033	23.255±0.52
Darshan	0.259±0.009	2.492±0.047	23.952±0.61
Score-P	0.270±0.008	2.588±0.051	24.883±0.63
EZIOTracer	0.263±0.012	2.785±0.055	26.844±0.68
DataCrumbs	0.252±0.004	2.422±0.067	23.278±0.52

monitoring of deep I/O stacks practical with minimal startup cost by limiting probes to only the functions used.

Table II compares application runtime under different profiling solutions. Across all event sizes, DataCrumbs adds minimal overhead and stays close to the base runtime, while Darshan adds a measurable increase (e.g., 0.697s for 1M events). Despite this, DataCrumbs captures far more events, 125M+ at 100K I/O events and 1.25B+ at 1M, versus Darshan's application-level-only statistics. Profiling overhead stays at about 0.1%, negligible for long-running HPC applications; for completeness, Score-P incurred 7% profiling overhead versus Darshan's 3%.

Table III evaluates tracing overhead, where DataCrumbs again stays close to the base runtime: at 100K events, Darshan DXT raises execution time to 2.624s versus DataCrumbs's 2.489s, despite DataCrumbs capturing more than 125M events. The same trend holds at 1M events, where DataCrumbs captures 1.25B+ events while incurring only 0.428s of overhead. Maximum tracing overhead is under 3%, versus 21% for Recorder and 26% for EZIOTracer, both substantially more expensive than DataCrumbs.

DataCrumbs captures orders of magnitude more events than Darshan at comparable or lower overhead, due to two design choices: event capture decoupled from storage, so the application is never blocked on writing events, and kernel-space capture, more efficient than user-space instrumentation. DataCrumbs also dynamically instruments 8,723 functions across Kernel, POSIX, MPI, and HDF5, providing a much broader view of I/O behavior. At 1M I/O events, DataCrumbs generates 128 MB of trace data per node (3 MB compressed), thanks to gzip compression, JSON Lines format, and string-hash deduplication. The user-space consumer sustained up to 12.3 M events/sec per node across all tested workloads without ring-buffer backpressure. DataCrumbs's per-node memory footprint, measured with and without it running, is 16 MB, corresponding to the kernel-to-user-space transfer buffer. The user-space consumer is pinned to a single core, with 12% CPU utilization from memory-bound event processing followed by file I/O, and DataCrumbs performs no network I/O.

TABLE III

TRACING OVERHEAD: DATACRUMBS CAPTURES >1000 MORE EVENTS WITH LOWER OVERHEAD THAN DARSHAN DXT, RECORDER, AND EZIOTRACER, WITH TYPICAL VARIABILITY < 1%.

Tool	10K Events / s	100K Events / s	1M Events / s
Base	0.252±0.001	2.419±0.033	23.255±0.52
Darshan DXT	0.269±0.01	2.624±0.047	25.350±0.61
Recorder	0.305±0.011	2.927±0.058	28.138±0.72
EZIOTracer	0.277±0.013	3.048±0.062	29.243±0.75
DataCrumbs	0.252±0.004	2.489±0.067	23.683±0.52

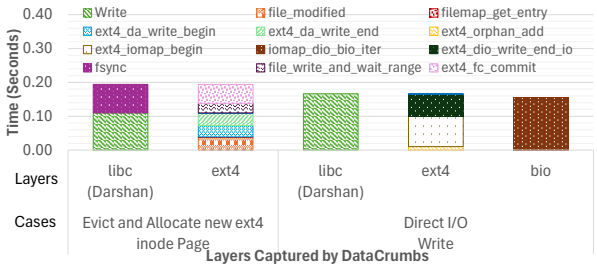


Fig. 6. DataCrumbs captures all layers and reveals the trade-offs between buffered and Direct I/O writes. While buffered writes are faster, fsync introduces significant overhead, making Direct I/O more efficient for data persistence.

Compared to EZIOTracer on buffered I/O with 32 threads, DataCrumbs achieves 0.1% vs. 15% profiling overhead (150×) and under 3% vs. 26% tracing overhead (refer to Tables II and III), due to libbpf ring buffers with asynchronous processing, as discussed in Section II.

C. Case Study: Benchmarks

DataCrumbs supports a range of I/O analysis use cases that reveal performance bottlenecks more deeply than API-level tools alone. Across all use cases, we apply Algorithm 1 with $\Delta = 1$ ms (chosen to match the kernel scheduling quantum and validated to capture all cross-layer associations in our workloads) to associate high-level API events (e.g., `H5Dwrite`, `H5Dread`) with their overlapping kernel-layer events, enabling cross-layer attribution. This section highlights two benchmark-driven examples: (1) analyzing performance differences between I/O paths and (2) explaining I/O behavior in a deep learning workload.

1) Analyzing Performance Differences between I/O Paths: For this use case, we switch IOR to the POSIX interface and compare buffered writes with fsync against Direct I/O writes. In profiler mode, DataCrumbs captures activity across `libc`, `ext4`, and the kernel block I/O layer.

Figure 6 presents runtime for both cases: buffered writes are initially 50% faster than Direct I/O due to in-memory ext4 caching, but fsync, which forces data to disk, offsets this advantage, making buffered writes with fsync 16% slower overall. A detailed ext4 breakdown shows buffered writes dominated by `file_modified` (35% of execution time) and `ext4_da_write_begin`/`ext4_da_write_end` (65%), while fsync adds overhead dominated by `file_write_and_wait_range` (33% of fsync time) and `ext4_fc_commit` (67%). Direct I/O instead bypasses caching via `ext4_orphan_add`

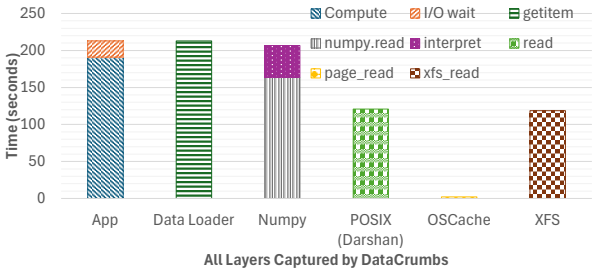


Fig. 7. DataCrumbs extracts all metrics from all layers to explain characteristics of deep learning workload.

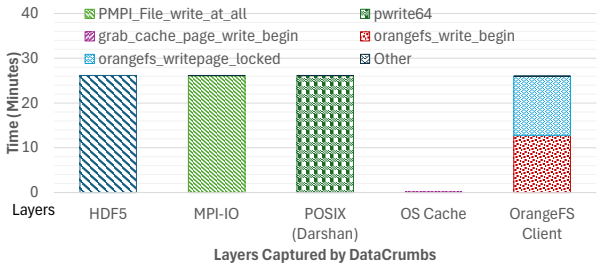


Fig. 8. DataCrumbs can explain that 99% of write performance of VPIC-IO is bound by OrangeFS performance.

(6%), `ext4_iomap_begin` (53%), and `ext4_dio_write_end_io` (39%), with kernel block-I/O contributing 93% of ext4's execution time. This highlights the buffered-vs-Direct-I/O trade-off: buffered writes are faster initially, but fsync's cost makes Direct I/O more efficient when immediate persistence is required, so applications needing persistence should prefer Direct I/O to avoid fsync's 16% write-back overhead. By capturing detailed execution across the stack, DataCrumbs provides the information needed for informed file-system tuning.

2) Explaining I/O Performance of a Deep Learning Workload: For DLIO, we analyze the Unet3D training workload described in Section V-A3. Figure 7 shows time spent across software layers while the main process overlaps computation with worker-side sample loading through DataLoader, NumPy, POSIX, and the file system. Although training runs on GPUs, we trace the host-side pipeline feeding them, leaving GPU-side I/O paths (e.g., GPUDirect Storage) to future work. Monitoring this workload with DataCrumbs costs under 0.3% overhead, with runtimes overlapping across repeated runs; this is substantially lower than the IOR overhead reported above because IOR is a C++ I/O benchmark that stresses the traced path directly, representing a worst case relative to DLIO's Python-heavy, compute-overlapped workload.

We observe that the main process spends only about 10% of its execution time waiting for I/O, so most I/O stalls overlap with computation and have limited impact on total training time. Worker processes handle data loading and preprocessing: the DataLoader layer spends 212 seconds on data samples, at an overall bandwidth of 232 GiB/s, 90% of which overlaps with computation, showing effective asynchronous prefetching. NumPy operations dominate the data-loading phase at 97% of its time cost, 76% (163 seconds) on I/O and the remainder converting binary data into Python arrays.

At the system-call level, the POSIX interface accounts for 120 seconds of data-reading cost, an effective bandwidth of 405 GiB/s, showing that the underlying system is capable of high-performance I/O, but some bandwidth is lost to Python-level overhead in NumPy's data interpretation, a gap between raw filesystem throughput and the effective bandwidth seen by the deep learning framework.Within the filesystem layer, 80% of the data is served directly from the OS cache at 235 GiB/s, while the remaining 20% is fetched from XFS at a much lower 1.2 GiB/s, a high cache hit rate indicating the workload benefits substantially from in-memory caching and needs no further I/O optimization.

This evaluation highlights DataCrumbs's ability to capture both synchronous and asynchronous I/O activity across multiple software layers, including Python libraries, C++ system calls, and the underlying filesystem stack. This visibility explains the observed bandwidths and shows that the workload is already well matched to the tested system. No further I/O optimization is needed; developer effort is better directed at reducing NumPy's binary-to-array conversion overhead (˜20% of DataLoader time) rather than storage tuning.

D. Case Study: Workflows

1) VPIC Simulation with BD-CATS Analysis: We next analyze the VPIC+BD-CATS producer-consumer workflow from Section V-A3, where VPIC checkpoints shared data and BD-CATS reads it for analysis [63], [64].

For the VPIC write path, the x-axis lists the I/O layers (HDF5, MPI-IO, POSIX, OS Cache, and OrangeFS client) and the y-axis reports time in minutes (Figure 8). The HDF5 `H5Dwrite_async` cost is 26.18 minutes, 99.7% of which is explained by the MPI-IO `PMPI_File_write_at_all` call (the remaining 0.3% falls into other MPI-IO work), and in turn 99.99% explained by POSIX `pwrite64`. At the kernel boundary, `pwrite64` splits into 0.13 minutes (0.5%) in the OS write-back cache and 25.97 minutes (99.5%) in OrangeFS client work, between `orangefs_write_begin` and `orangefs_writepage_locked`; the OS-cache share represents data buffered for future reads, while the full `pwrite64` cost is paid in the OrangeFS client path. Within OrangeFS, the page-locking path dominates, indicating a client-side bottleneck, so since VPIC uses large, page-aligned writes, we recommend enabling Direct-IO for checkpointing to bypass this page-locking overhead. This also shows why tools that stop at POSIX (e.g., Darshan) miss the filesystem-client bottleneck and cannot explain this low-level issue.

For BD-CATS reads, the x-axis again enumerates the I/O layers and the y-axis reports time in minutes (Figure 9). The HDF5 `H5Dread_async` time is 3.22 minutes, 99.7% explained by MPI-IO `PMPI_File_read_at`, in turn 99.99% explained by POSIX `pread64`. Below POSIX, `pread64` divides into OS cache `read_page` (1.35 minutes, 42%) and OrangeFS client revalidation (1.85 minutes, 58%), showing that a large fraction of reads benefit from write-back cache locality; as with VPIC, tools that stop at POSIX (e.g., Darshan) cannot attribute this split, obscuring how much

latency comes from cached reads versus OrangeFS access. Increasing the OS page cache size or pre-staging reused datasets would therefore improve BD-CATS read performance by raising the 42% cache hit rate.

2) Montage Workflow: We also analyze the Montage workflow, a six-stage FITS-to-PNG mosaic pipeline for survey NGC 3372, using the same input data while scaling execution from 8 to 32 nodes. Across all scales and instrumented layers, DataCrumbs has an overhead of around 0.52%, showing that monitoring cost remains effectively constant with scale.

For Montage, the x-axis lists the I/O layers and the y-axis reports time in minutes (Figure 10). The POSIX layer shows a read-dominated workload (37.87 minutes read vs. 0.61 minutes write), difficult to understand because the workflow contains multiple compute stages and an implicit producer-consumer pattern; this behavior stays consistent across the 8-, 16-, and 32-node runs since the workflow processes the same dataset at each scale, and monitoring overhead likewise stays unchanged, averaging 0.52% across all scales and layers, rising to 0.9% when the calculation is restricted to instrumented, traced phases and excludes untraced compute stages. Below POSIX, 15.48 minutes (41%) of read time is served by the OS cache, indicating strong locality, while the remaining 22.35 minutes (59%) comes from the OrangeFS client. Within OrangeFS, read-ahead is critical: `orangefs_readahead` accounts for 11.55 minutes, roughly half of OrangeFS read time, with `orangefs_file_read_iter` the rest, so increasing the read-ahead prefetch size should improve Montage read performance. The trace for the 32-node run, our largest case, captured 1.5B events, totaling 142 MB on disk after gzip compression, confirming the format remains compact at scale. **Targeted Optimization:** Based on this observation, we enlarged the OrangeFS client's read-ahead window and added sequential and prefetch access hints to Montage's read path [65], [66]. Re-tracing the tuned run confirms the targeted effect: time in `orangefs_readahead` rises while flush-triggered reads fall, and overall POSIX time drops by 1.35×, showing that DataCrumbs's attribution can directly guide and verify a targeted optimization.

VI. Conclusions

Our evaluation highlights four main results. First, DataCrumbs flexibly attaches to multiple I/O software components, instrumenting 8,723 functions at only 78 ms of startup overhead. Second, it efficiently profiles and traces both user-

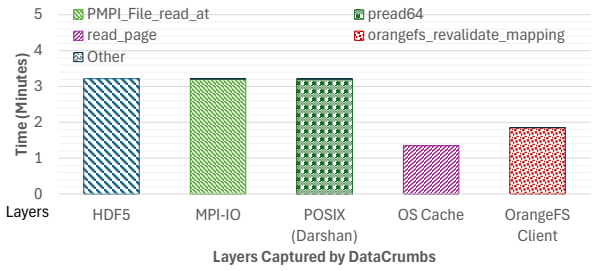


Fig. 9. DataCrumbs can explain that 42% of reads are served by OS Cache and rest by OrangeFS for BD-CATS.

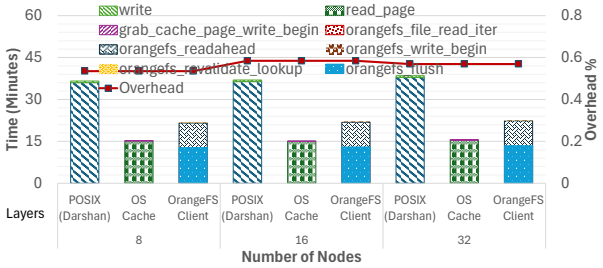


Fig. 10. DataCrumbs shows that about 41% of Montage reads are served by the OS cache, with the rest handled by OrangeFS, while maintaining a stable 0.52% average overhead from 8 to 32 nodes; "(Darshan)" marks the layer also visible to Darshan.

and kernel-space I/O operations at just 0.1% profiling overhead and under 3% tracing overhead (via decoupled, kernel-space event capture with asynchronous ring-buffer transfer), versus 3% for Darshan profiling and ∼9% for Darshan DXT tracing. Third, its time-windowed event-association algorithm reduces attribution cost from $\mathcal{O}(n^2)$ to $\mathcal{O}((n/t)^2)$ by partitioning events into independent time windows, enabling root-cause analysis of I/O variability, cross-layer interaction effects, and within-layer bottlenecks. Fourth, across four use cases (IOR, DLIO, VPIC+BD-CATS, Montage), DataCrumbs exposed actionable, cross-layer insights unavailable to user-space-only tools; at 32-node scale with Montage it maintained a stable 0.52% overhead, generating only 142 MB of compressed trace data from 1.5B events, and revealed that 41% of POSIX reads were served from the OS cache while OrangeFS read-ahead dominated the rest. Overall, DataCrumbs makes explainable I/O analysis practical for production HPC storage systems.

In the future, we plan to deploy DataCrumbs on Livermore Computing (LC) at LLNL, Argonne National Laboratory, and Oak Ridge National Laboratory testbeds, extend its support to GPU-local storage middleware, and enable cross-node runtime correlation (currently performed offline). Because our use cases already show workloads well-served by the OS page cache (e.g., 41% of Montage and 42% of BD-CATS reads), we also plan to explore DataCrumbs-driven targeted optimization, using the tool's cross-layer attribution to automatically recommend and, eventually, apply layer-specific tuning such as Direct-IO or cache-sizing changes.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 24-SI-005 (LLNL-CONF-2004682). Also, this work was partly developed and tested on computing resource at Gnosis Research Center at Illinois Tech.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] S. Neuwirth and A. K. Paul, "Parallel I/O Evaluation Techniques and Emerging HPC Workloads: A Perspective," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 671–679.

[2] Z. Zhu, L. Derstroff, and S. Neuwirth, "Benchmarking Darshan and Recorder for HPC I/O Profiling and Tracing," in *2025 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, 2025, pp. 1–6.

[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)*, Nov. 2016, pp. 9–17.

[4] ALCF, "ALCF Public Data: Argonne Leadership Computing Facility," Jan. 2020. [Online]. Available: https://reports.alcf.anl.gov/data/mira.html

[5] NERSC, "Darshan I/O Profiler - NERSC Documentation," Jul. 2024. [Online]. Available: https://docs.nersc.gov/tools/performance/darshan/

[6] OLCF, "Data Storage and Transfers — OLCF User Documentation," Jan. 2025. [Online]. Available: https://docs.olcf.ornl.gov/data/index.html#darshan-runtime-and-i-o-profiling

[7] H. Devarajan, L. Pottier, K. Velusamy, H. Zheng, I. Yildirim, O. Kogiou, W. Yu, A. Kougkas, X.-H. Sun, J. S. Yeom, and K. Mohror, "DFTracer: An Analysis-Friendly Data Flow Tracer for AI-Driven Workflows," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. Atlanta, GA: IEEE, Jun. 2024.

[8] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient Parallel I/O Tracing and Analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2020, pp. 1–8.

[9] C. Xu, S. Snyder, O. Kulkarni, V. Venkatesan, P. Carns, S. Byna, R. Sisneros, and K. Chadalavada, "DXT: Darshan eXtended Tracing."

[10] M. I. Naas, F. Trahay, A. Colin, P. Olivier, S. Rubini, F. Singhoff, and J. Boukhobza, "EZIOTracer: unifying kernel and user space I/O tracing for data-intensive applications," in *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, 2021, pp. 1–11.

[11] J. L. Bez, H. Ather, and S. Byna, "Drishti: Guiding End-Users in the I/O Optimization Journey," in *2022 IEEE/ACM International Parallel Data Systems Workshop (PDSW)*, Nov. 2022, pp. 1–6.

[12] I. Yildirim, H. Devarajan, A. Kougkas, X.-H. Sun, and K. Mohror, "IOMax: Maximizing Out-of-Core I/O Analysis Performance on HPC Systems," in *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA*, Nov. 2023.

[13] G. K. Lockwood, N. J. Wright, S. Snyder, P. Carns, G. Brown, and K. Harms, "TOKIO on ClusterStor: Connecting Standard Tools to Enable Holistic I/O Performance Analysis," 2018. [Online]. Available: https://escholarship.org/uc/item/8j14j182

[14] H. Devarajan and K. Mohror, "Extracting and characterizing I/O behavior of HPC workloads." Heidelberg, Germany: IEEE, Sep. 2022.

[15] C. Wang, K. Mohror, and M. Snir, "File System Semantics Requirements of HPC Applications," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '21. New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 19–30. [Online]. Available: https://doi.org/10.1145/3431379.3460637

[16] N. Monnier, J. Lofstead, M. Lawson, and M. Curry, "Profiling platform storage using io500 and mistral," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*. IEEE, 2019, pp. 60–73.

[17] N. Siva, "1000 genomes project," *Nature Biotechnology*, vol. 26, no. 3, pp. 256–257, Mar. 2008. [Online]. Available: https://github.com/pegasus-isi/1000genome-workflow

[18] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Kärnä, D. Moise, S. J. Pennycook, K. Maschhoff, J. Sewall, N. Kumar, S. Ho, M. F. Ringenburg, P. Prabhat, and V. Lee, "CosmoFlow: Using Deep Learning to Learn the Universe at Scale," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2018, pp. 819–829. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8665771

[19] M. Rynge, G. Juve, J. Kinney, J. Good, B. Berriman, A. Merrihew, and E. Deelman, "Producing an Infrared Multiwavelength Galactic Plane Atlas Using Montage, Pegasus, and Amazon Web Services," *Annual Astronomical Data Analysis Software and Systems (ADASS)*, vol. 485, p. 211, May 2014, conference Name: Astronomical Data Analysis Software and Systems XXIII ADS Bibcode: 2014ASPC..485..211R. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2014ASPC..485..211R

[20] H.-D. Do, V. Hayot-Sasson, R. F. Da Silva, C. Steele, H. Casanova, and T. Glatard, "Modeling the Linux page cache for accurate simulation of data-intensive applications," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 398–408.

[21] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux symposium*, vol. 2. Citeseer, 2007, pp. 21–33.

[22] D. Oh, "System level jitter characterization of high speed I/O systems," in *2012 IEEE International Symposium on Electromagnetic Compatibility*. IEEE, 2012, pp. 173–178.

[23] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, ser. IOPADS '99. New York, NY, USA: Association for Computing Machinery, May 1999, pp. 23–32.

[24] "Open MPI IO Tuning," Mar. 2025. [Online]. Available: https://github.com/open-mpi/ompi/blob/main/docs/tuning-apps/mpi-io.rst

[25] P. Carns, J. Kunkel, K. Mohror, and M. Schulz, "Understanding I/O Behavior in Scientific and Data-Intensive Computing (Dagstuhl Seminar 21332)," *Dagstuhl Reports*, vol. 11, no. 7, pp. 16–75, 2021.

[26] J. Garlick, "LMT3 - Lustre Monitoring Tool," Jan. 2025, original-date: 2011-03-04T22:06:14Z. [Online]. Available: https://github.com/LLNL/lmt

[27] IBM, "mmpmon," Aug. 2024. [Online]. Available: https://www.ibm.com/docs/en/storage-scale/5.2.2?topic=reference-mmpmon-command

[28] A. K. Paul, R. Chard, K. Chard, S. Tuecke, A. R. Butt, and I. Foster, "Fsmonitor: Scalable file system monitoring for arbitrary storage systems," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–11.

[29] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, Nov. 2008, pp. 1–12.

[30] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Berlin, Heidelberg: Springer, 2012, pp. 79–91.

[31] C. Wang, I. Yildirim, H. Devarajan, K. Mohror, and M. Snir, "Recorder: Comprehensive Parallel I/O Tracing and Analysis," *arXiv preprint arXiv:2501.04654*, 2025.

[32] Tetragon, "Tetragon - eBPF-based Security Observability and Runtime Enforcement," May 2024. [Online]. Available: https://tetragon.io/

[33] "Process Monitor for Linux," Mar. 2025, original-date: 2020-07-10T22:58:07Z. [Online]. Available: https://github.com/microsoft/ProcMon-for-Linux

[34] 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*, ser. SSDBM '25. New York, NY, USA: Association for Computing Machinery, Jun. 2025, pp. 1–6. [Online]. Available: https://dl.acm.org/doi/10.1145/3733723.3733741

[35] B. Gbadamosi, L. Leonardi, T. Pulls, T. Høiland-Jørgensen, S. Ferlin-Reiter, S. Sorce, and A. Brunström, "The eBPF Runtime in the Linux Kernel," Oct. 2024, arXiv:2410.00026 [cs]. [Online]. Available: http://arxiv.org/abs/2410.00026

[36] M. Deokar, J. Men, L. Castanheira, A. Bhardwaj, and T. A. Benson, "An Empirical Study on the Challenges of eBPF Application Development," in *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, ser. eBPF '24. New York, NY, USA: Association for Computing Machinery, Aug. 2024, pp. 1–8. [Online]. Available: https://dl.acm.org/doi/10.1145/3672197.3673429

[37] "Cilium - Cloud Native, eBPF-based Networking, Observability, and Security," 2025. [Online]. Available: https://cilium.io

[38] D. Soldani, P. Nahi, H. Bour, S. Jafarizadeh, M. F. Soliman, L. Di Giovanna, F. Monaco, G. Ognibene, and F. Risso, "eBPF: A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond)," *IEEE Access*, vol. 11, pp. 57 174–57 202, 2023, conference Name: IEEE Access. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10138542

[39] A. Khichane, "Diagnostic of performance by data interpretation for 5G cloud native network functions," phdthesis, Université Paris-Saclay, Mar. 2024. [Online]. Available: https://theses.hal.science/tel-04982832

[40] C. Cassagnes, L. Trestioreanu, C. Joly, and R. State, "The rise of eBPF for non-intrusive performance monitoring," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, Apr. 2020, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9110434

[41] I. Visor, "BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more," Apr. 2025, original-date: 2015-05-01T19:52:32Z. [Online]. Available: https://github.com/iovisor/bcc

[42] A. Robertson, "bpftrace: High-level tracing language for Linux," Apr. 2025, original-date: 2018-08-31T04:34:44Z. [Online]. Available: https://github.com/bpftrace/bpftrace

[43] Y. Zheng, T. Yu, Y. Yang, Y. Hu, X. Lai, and A. Quinn, "bpftime: userspace eBPF Runtime for Uprobe, Syscall and Kernel-User Interactions," Dec. 2023, arXiv:2311.07923 [cs]. [Online]. Available: http://arxiv.org/abs/2311.07923

[44] bpf, "BPF CO-RE - eBPF Docs," Aug. 2025. [Online]. Available: https://docs.ebpf.io/concepts/core/

[45] The HDF Group, "Hierarchical Data Format, version 5," 1997-2022, https://www.hdfgroup.org/HDF5/.

[46] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel NetCDF: A High-Performance Scientific I/O Interface," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 39. [Online]. Available: https://doi.org/10.1145/1048935.1050189

[47] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Input/output apis and data organization for high performance scientific computing," in *2008 3rd Petascale Data Storage Workshop*. IEEE, 2008, pp. 1–6.

[48] M. J. Brim, A. T. Moody, S.-H. Lim, R. Miller, S. Boehm, C. Stanavige, K. M. Mohror, and S. Oral, "UnifyFS: A User-level Shared File System for Unified Access to Distributed Local Storage," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2023, pp. 290–300.

[49] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 219–230.

[50] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," 2002. [Online]. Available: https://shorturl.at/yLV6P

[51] P. Braam, "The Lustre Storage Architecture," *arXiv:1903.01955 [cs]*, Mar. 2019, arXiv: 1903.01955. [Online]. Available: http://arxiv.org/abs/1903.01955

[52] F. Isaila, J. Garcia, J. Carretero, R. Ross, and D. Kimpe, "Making the case for reforming the I/O software stack of extreme-scale systems," *Advances in Engineering Software*, vol. 111, pp. 26–31, Sep. 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0965997816301740

[53] H. Devarajan, G. Heber, and K. Mohror, "H5Intent: Autotuning HDF5 With User Intent," *IEEE Transactions on Parallel and Distributed Systems*, vol. 36, no. 2, pp. 108–119, 2025.

[54] C. Wang, K. Mohror, and M. Snir, "Formal Definitions and Performance Comparison of Consistency Models for Parallel File Systems," *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[55] H. Devarajan, "DataCrumbs," Apr. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.15257875

[56] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The Spack package manager: bringing order to HPC software chaos," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, Nov. 2015, pp. 1–12. [Online]. Available: https://doi.org/10.1145/2807591.2807623

[57] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2010, pp. 1–11.

[58] S. Byna, Q. Koziol, V. Vishwanath, J. Soumagne, H. Tang, J. Mu, B. Dong, R. A. Warren, F. Tessier, T. Wang, and J. Liu, "Proactive Data Containers (PDC): An Object-centric Data Store for Large-scale Computing Systems," *AGU Fall Meeting Abstracts*, vol. 2018, pp. IN34B–09, Dec. 2018.

[59] M. Rocklin, "Dask: Parallel Computation with Blocked algorithms and Task Scheduling," Jan. 2015, pp. 126–132.

[60] D. Eschweiler, M. Wagner, M. Geimer, Kn&#252, A. Pfer, W. E. Nagel, and F. Wolf, "Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries," in *Applications, Tools and Techniques on the Road to Exascale Computing*. IOS Press, 2012, pp. 481–490. [Online]. Available: https://ebooks.iospress.nl/doi/10.3233/978-1-61499-041-3-481

[61] "Ares User Guide | Gnosis Research Center," Chicago. [Online]. Available: https://grc.iit.edu/docs/ares/ares-user-guide

[62] J. Kunkel, "HPC IO Benchmark v 4.0.0," Apr. 2025, original-date: 2015-06-23T15:56:39Z. [Online]. Available: https://github.com/hpc/ior/releases/tag/4.0.0

[63] N. P. Tan, R. B. F. Bird, G. Chen, S. V. Luedtke, B. J. Albright, and M. Taufer, "Analysis of Vector Particle-In-Cell (VPIC) memory usage optimizations on cutting-edge computer architectures," *Journal of Computational Science*, vol. 60, Jan. 2022, institution: Los Alamos National Lab. (LANL), Los Alamos, NM (United States) Number: LA-UR-21-31487. [Online]. Available: https://www.osti.gov/pages/biblio/1844166

[64] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, Prabhat, and P. Dubey, "BD-CATS: big data clustering at trillion particle scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, Nov. 2015, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/2807591.2807616

[65] "blockdev(8): Call block device ioctls - Linux man page." [Online]. Available: https://man7.org/linux/man-pages/man8/blockdev.8.html

[66] "fadvise(2): Give advice about file access - Linux man page." [Online]. Available: https://linux.die.net/man/2/fadvise