Drilling Down I/O Bottlenecks with Cross-layer I/O Profile Exploration

Hammad Ather*†§, Jean Luca Bez†§, Yankun Xia‡, Suren Byna‡†
*University of Oregon, †Lawrence Berkeley National Laboratory, ‡The Ohio State University
*hather@uoregon.edu, †jlbez@lbl.gov, ‡xia.720@buckeyemail.osu.edu, byna.1@osu.edu

***Abstract*—I/O performance monitoring tools such as Darshan and Recorder collect I/O-related metrics on production systems and help understand the applications' behavior. However, some gaps prevent end-users from seeing the whole picture when it comes to detecting and drilling down to the root causes of I/O performance slowdowns and where those problems originate. These gaps arise from limitations in the available metrics, their collection strategy, and the lack of translation to actionable items that could advise on optimizations. This paper highlights such gaps and proposes solutions to drill down to the source code level to pinpoint the root causes of I/O bottlenecks scientific applications face by relying on cross-layer analysis combining multiple performance metrics related to I/O software layers. We demonstrate with two real applications how metrics collected in high-level libraries (which are closer to the data models used by an application), enhanced by source-code insights and natural language translations, can help streamline the understanding of I/O behavior and provide guidance to end-users, developers, and supercomputing facilities on how to improve I/O performance. Using this cross-layer analysis and the heuristic recommendations, we attained up to 6.9× speedup from run-as-is executions.**

***Index Terms*—I/O bottleneck, source code analysis, root causes**

I. Introduction

High-Performance Computing (HPC) systems provide a multi-layered I/O software stack to support serial and parallel data access from scientific applications, as Fig. 1 illustrates. This approach hides the complexities and particularities of the underlying layer behind abstractions, exposing (at the top) data models that closely map to the application's data abstractions rather than the internals of a storage system (at the bottom).

Hence, it becomes natural that data transformations reshape an application's access pattern while I/O requests traverse the I/O stack, passing through high-level and middleware I/O libraries, undergoing various optimization steps, until reaching a parallel file system (PFS) [1]. However, such transformations are often transparent to end-users and application developers, who only feel the effects of their success or overheads. While some I/O optimizations expose tunable parameters to accommodate distinct workload characteristics, others rely on source-code changes to apply best practices in data representation, data movement, and storage layout. Hence, visualizing and understanding those transformations and where poorly performing accesses originate in the source code (caused by either misusing each layer or by the transformations) becomes paramount to determine an appropriate course of action and correct optimization techniques for each situation.

Furthermore, as novel workloads began to require HPC resources and enter supercomputer facilities [2]–[5], they will need to adhere to best practices to fully harness performance from the existing I/O stack. Hence, understanding how they will run on such systems, and how their data access model will seamlessly or not map to this cross-layered structure will become vital to enable faster scientific discovery. Such problem is still faced by traditional scientific simulation applications when running for the first time in a new large-scale platform. For instance, despite previous studies [6]–[9] indicating that small requests have a negative impact on I/O performance, widespread usage of small I/O requests is still observed today in applications across science domains [10], [11]. To complicate matters further, optimization techniques such as collective buffering and data sieving [12], which have been proposed decades ago and demonstrate performance gains are not yet adopted in cases it should have been. Automatic tuning mechanisms [13]–[15] have also been proposed to tweak and adapt some of the techniques to best suit the application, and though they have been successful in some cases, they can be expensive to train and limited in what they can do (as they will only affect exposed options). Moreover, some changes are intrinsic to the way the application accesses its data, and modifying that requires knowledge of the impact of those design choices in the HPC I/O stack.

From the existing profiling tools used in the HPC context to understand application behavior [16]–[19], some, such as Darshan [20] and Recorder [21], specialize in collecting I/O-related performance metrics and traces. Using different approaches to collect data and distinct formats to store their observations, they seek to provide the means to characterize the I/O behavior. Each has its potential and limitations, as further discussed in Section II, but both can aid in uprooting the causes of I/O performance issues. However, the final feedback loop back to end-users and developers is not there, leaving the communication and interpretation of the metrics and findings restricted to I/O experts (researchers, system admins), making them incomprehensible to many users [22].

Recent efforts attempted to fill the gap between the collected metrics and their interpretations and translation into optimizations. One such tool is DXT Explorer [23], an interactive web-based trace analysis tool. This tool tries to connect the dots

§These authors contributed equally to this work.

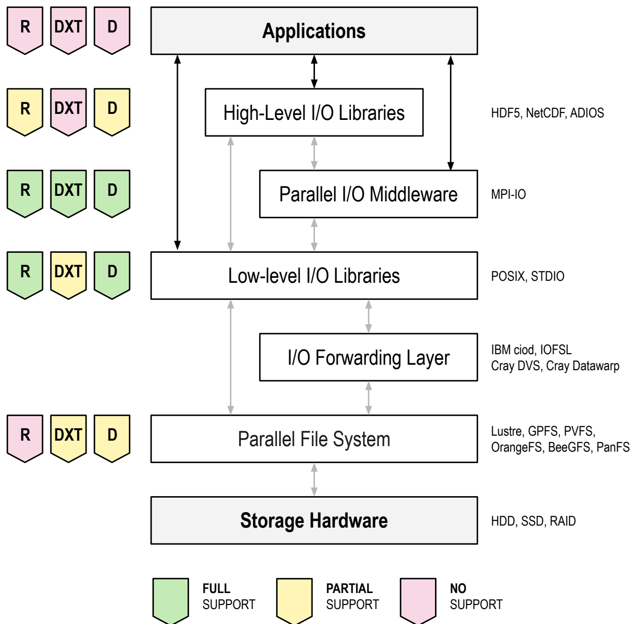


Fig. 1. Traditional parallel I/O stack deployed in production scale supercomputers and the coverage of common I/O profiling and tracing tools. (R) Recorder, (D) Darshan, and (DXT) Darshan eXtended Tracing. For those maked with partial support, the feature has some caveats or limitations in what type of metric is collected or its usage.

between I/O bottleneck detection and optimization through interactive visualizations. It provides a detailed view of the I/O behavior of the application exclusively from DXT traces in the form of interactive visualizations with the ability to zoom in and out in regions of interest. Similarly, *recorder-viz* is a solution crafted for Recorder [21] traces. Drishti [24], [25], on the other hand, tries to identify I/O performance pitfalls based on a set of heuristic-based triggers and provide insights on optimizations in the form of a report. Although these tools are one step in the right direction, they still lack the capability to do cross-layer I/O analysis by combining I/O metrics from different sources and drilling down to the root cause in source code, closing this feedback loop.

The "*Understanding I/O Behavior in Scientific and Data-Intensive Computing*" report [22], which brought together computer scientists worldwide to survey how I/O workloads are measured and analyzed on HPC systems, highlights some existing gaps in methodology and technology to advance HPC productivity. Among them, some key challenges are low-fidelity acquisition, lack of hierarchical scope, incompatible formats, the complexity of analysis, and lack of a standard.

In this paper, we work towards solving some of those challenges. We seek to explore how different sources of I/O metrics can be abstracted and harnessed to provide actionable insights to end-users, drilling down to the source-code causes (where appropriate) and reducing the complexity of analysis. We also highlight the existing gaps in metric collection and mismatching representations and discuss tradeoffs and overheads. Our work lays out the foundations for an end-to-end solution to uproot I/O performance problems in their origin.

The remainder of the paper is organized as follows. In Section II, we examine different sources of I/O-related metrics, their opportunities, and limitations. Section III discusses the design choices to drill down to the root causes in the source code. Section IV approaches how metrics from high-level libraries can enhance the understanding of I/O behavior. We demonstrate the feedback to end-users and developers in Section V. Related work is covered in Section VI. We conclude the paper in Section VII and discuss future R&D.

II. Data Sources and Limitations

In this section, we discuss some of the existing tools that can collect I/O-related metrics across the HPC I/O stack, their support, and limitations. We also highlight how they can used to infer behavior and identify I/O performance bottlenecks.

A. Darshan

Darshan [20] is a tool deployed on several large-scale HPC facilities to collect I/O-related performance metrics and aid in understanding how applications use such a complex stack. Darshan provides an efficient, transparent, and compact runtime instrumentation of many standard I/O interfaces (POSIX, MPI-I/O, STDIO). It also includes additional modules (e.g., DXT, HDF5, Lustre), that enhance the tool's capabilities based on the application's demands. Fig. 2 depicts the extensible format used by Darshan to store these profiling metrics.

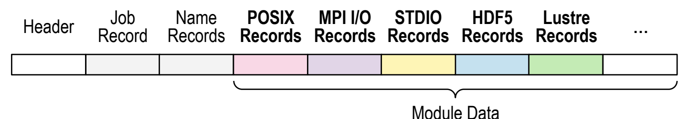


Fig. 2. Darshan format to store I/O metrics from multiple modules/sources.

B. Darshan DXT

Darshan eXtended Tracing (DXT) [26] extends Darshan by providing fine-grain traces of the I/O of the application, which can be used to get an in-depth view of the I/O behavior. These traces are recorded only for the POSIX and MPI-IO interfaces and include details such as the file, offset, length, start and end times, and the rank issuing that request. DXT traces are often used for offline post-mortem in-depth analysis to identify I/O issues. Although the overhead of the DXT module is minimal in many cases, it can be high (in time and space) for unoptimized applications. That is why the DXT module is turned off by default on production systems.

C. Recorder

Recorder [21] is another application-level tracing tool focused on I/O. It captures function calls at multiple levels of the I/O stack, including HDF5, MPI-IO, and POSIX I/O. Like Darshan, Recorder does not require an application to be recompiled to be profiled/traced, as this is activated by setting the `LD_PRELOAD` environment variable. Recorder, however, exposes some fine-grain control regarding which levels are traced, but it does not yet support other high-level libraries, such as NetCDF. Differently from Darshan, the traces and metrics collected by Recorder are stored in a Recorder-specific

status	**start**	**end**	**function**	arg_1	...	arg_n
1 byte	4 bytes	4 bytes	1 byte	variable size		

Fig. 3. Recorder format representation to store I/O traces.

tracing format using a format-aware compression algorithm, yielding a directory containing three files per rank plus a metadata file, instead of a single self-contained file.

Fig. 3 depicts the log format used by Recorder. Each record is identified as either a compressed or uncompressed trace, and the status byte distinguishes between the two. The start and end bytes signify the function execution period. The following function byte is an integer representing the function name. The remaining bytes represent variable-length function arguments. The compression process keeps a slicing window that retains a subset of recent traces. For each new trace, it checks whether a trace exists with the same function call and at least one matching argument. If so, the status byte of the new trace is configured with the first bit set to 1, and the following bits indicating the indices of arguments that are different. Additionally, the function byte stores the relative location to that similar trace instead. The trace is compressed by only keeping the difference.

D. High-Level Libraries

Darshan and Recorder have partial support for collecting high-level library I/O counters, as depicted by Fig. 1. Darshan has specific modules to capture HDF5 and PnetCDF performance-related counters, but it does not yet have support to collect traces through its extended tracing module (DXT).

Regarding HDF5, an application must use the same HDF5 library version that Darshan or Recorder was compiled against. On the one hand, Darshan can capture aggregated metrics covering the H5F (files) and H5D (dataset) HDF5 APIs. On the other hand, Recorder intercepts more HDF5 APIs, grabbing detailed information from files, groups, datasets, attributes, objects, links, and property lists. Furthermore, Recorder does not yet support other high-level libraries, such as PnetCDF. However, Darshan can capture aggregated metrics for two key PnetCDF abstractions (files and variables) but no traces.

Having information from high-level libraries is important to understand how the data abstractions that are natural to an application domain, map into I/O libraries and how those interface with the underlying layers, commonly triggering transformation. Since both HDF5 and PnetCDF expose a rich user-defined metadata API, this gap in metric collection and coverage of the I/O stack can potentially hide application-level metadata-related issues. We demonstrate this in Section V.

E. Parallel File Systems

In HPC systems, applications read and write data to shared Parallel File Systems (PFS), which provides a globally persistent storage infrastructure and a global namespace. A PFS is comprised of two types of servers with distinct roles: the data servers and the metadata servers. The latter handles details about the files themselves (e.g., sizes and permissions) and their location in the system. Lustre [27], [28] and IBM Spectrum Scale (previously known as GPFS) [29] are the ones commonly deployed in HPC facilities.

Darshan can collect basic information about Lustre, covering the number of OSTs (Object Storage Targets) and MDTs (Metadata Targets) in the system. When such a module is available and enabled, Darshan also collects striping information (size, offset, and width) for each file, which could hint at how the POSIX layer accesses the file system. However, Darshan does not capture anything specific for the Spectrum Scale.

Nonetheless, other sources of metrics could be combined to complete the cross-level view of how requests reach the file system. In the case of Lustre, some metrics are exposed by vendors, e.g., in ClusterStor, while others rely on open-source solutions such as *collectl-lustre* [30] and Lustre Monitoring Tool (LMT) [31]. Besides each of these solutions having their own format, correlating these file system metrics (collected as cumulative counters in time-based intervals) with job or application metrics is very complex. The complexity also arises from network-related factors and gaps while associating metrics from upper layers without losing the necessary context for I/O-related analysis. Though we acknowledge that they should be part of this cross-layer exploration, due to such complexity, we leave this problem as future work.

III. Drilling Down to the Source-Code

While end-users may be aware of where their input/output (I/O) calls are being made, the real challenge arises due to the multiple transformations that an I/O request undergoes while traversing the HPC I/O stack. This makes it difficult to identify the root cause of any performance issues (i.e., what in the application code ended up impairing performance). Consequently, what a user thinks their application is doing might not be the case. This leads to several known I/O performance problems.

Source code analysis is an important technique to dig deeper into the vulnerabilities and bottlenecks in an HPC application. Using different source code analysis methodologies, we can drill down to the source of the I/O bottlenecks, making it easier to optimize the application. Source code analysis in HPC I/O applications can have some challenges as we deal with vast amounts of data, which can result in added overhead.

This section presents a framework for source code analysis in HPC I/O applications prototyped inside Darshan. We enhance the Darshan logs with additional information related to the I/O source code of HPC applications that *Drishti* uses to provide the exact line numbers where optimizations need to be made. We also discuss the design choices we made while developing this framework and the experiments we conducted to choose a suitable library for source code analysis. As mentioned, this framework is currently developed inside Darshan, but it can be integrated into other I/O profiling and tracing tools such as Recorder.

A. Unveiling the Source-Code

A backtrace is a list of function calls that are currently active in a program. Though external debuggers are often

```
1  /darshan/lib/libdarshan.so(dxt_posix_write+0x118) [0x7f4afdbabc58]
2  /darshan/lib/libdarshan.so(pwrite+0x19d) [0x7f4afdb9761d]
3  /opt/cray/pe/lib64/libmpi_gnu_91.so.12(+0x26007f3) [0x7f4afcae37f3]
4  /opt/cray/pe/lib64/libmpi_gnu_91.so.12(+0x2605cec) [0x7f4afcae8cec]
5  /opt/cray/pe/lib64/libmpi_gnu_91.so.12(+0x25d60d9) [0x7f4afcab90d9]
6  /opt/cray/pe/lib64/libmpi_gnu_91.so.12(PMPI_File_write_at_all+0x21)[0x7f4afcabaa11]
7  /darshan/lib/libdarshan.so(MPI_File_write_at_all+0xbb) [0x7f4afdbaff1b]
8  /hdf5/lib/libhdf5.so.310(+0x38913c) [0x7f4afd65e13c]
9  /hdf5/lib/libhdf5.so.310(H5FD_write+0x68) [0x7f4afd41cda8]
10 /hdf5/lib/libhdf5.so.310(H5F__accum_write+0x194) [0x7f4afd3f6794]
11 /hdf5/lib/libhdf5.so.310(H5PB_write+0x6cb) [0x7f4afd50f3fb]
12 /hdf5/lib/libhdf5.so.310(H5F_shared_block_write+0x30) [0x7f4afd401ce0]
13 /hdf5/lib/libhdf5.so.310(H5D__mpio_select_write+0x22) [0x7f4afd65c402]
14 /hdf5/lib/libhdf5.so.310(+0x37a8d3) [0x7f4afd64f8d3]
15 /hdf5/lib/libhdf5.so.310(+0x37aaa5) [0x7f4afd64faa5]
16 /hdf5/lib/libhdf5.so.310(+0x3854e2) [0x7f4afd65a4e2]
17 /hdf5/lib/libhdf5.so.310(H5D__collective_write+0x9) [0x7f4afd65c4c9]
18 /hdf5/lib/libhdf5.so.310(H5D__write+0x166) [0x7f4afd3c0586]
19 /hdf5/lib/libhdf5.so.310(H5VL__native_dataset_write+0x8f) [0x7f4afd61dfdf]
20 /hdf5/lib/libhdf5.so.310(H5VL_dataset_write_direct+0x81) [0x7f4afd6085d1]
21 /hdf5/lib/libhdf5.so.310(+0xb5ef0) [0x7f4afd38aef0]
22 /hdf5/lib/libhdf5.so.310(H5Dwrite+0x9f) [0x7f4afd38e20f]
23 /darshan/lib/libdarshan.so(H5Dwrite+0xd7) [0x7f4afdbbc757]
24 /h5bench/bin/h5bench_e3sm() [0x6c986b]
25 /h5bench/bin/h5bench_e3sm() [0x457c4b]
26 /h5bench/bin/h5bench_e3sm() [0x454e87]
27 /h5bench/bin/h5bench_e3sm() [0x452947]
28 /h5bench/bin/h5bench_e3sm() [0x451f1c]
29 /lib64/libc.so.6(__libc_start_main+0xef) [0x7f4af9fba24d]
30 /h5bench/bin/h5bench_e3sm() [0x4525da]
```

Fig. 4. Sample backtrace from an HDF5 application monitored with Darshan.

used to inspect a backtrace, there are scenarios where it is useful to obtain a backtrace programmatically from within a program. For this purpose, the Standard C library exposes the `execinfo.h` header file, which declares functions that obtain and manipulate backtraces[§].

The `backtrace()` function obtains the backtrace for the calling thread as a list of pointers and places those in a buffer. Each entry in the buffer contains one return address per stack frame, limited by how deep one wants to investigate. It is important to notice that this approach has some caveats. Particular compiler optimization may interfere with obtaining a valid backtrace. Furthermore, inline functions do not have a stack frame, tail call optimizations replace one stack frame with another, and frame pointer elimination will stop the backtrace from correctly interpreting the contents of the stack.

Once the addresses from `backtrace()` call are available, `backtrace_symbols()` can be used to get the symbolic representation of the addresses in the form of a string. The representation of each address has the function name, the hexadecimal offset, and the actual hexadecimal return address. Fig. 4 shows a sample backtrace of a write call from an HPC application that uses the HDF5 I/O library and Darshan. It is possible to see the symbolic representation of the addresses of the call chain of this request. However, that backtrace contains no source-level information, such as line or function names. Nevertheless, it is possible to get this data by employing some libraries and tools such as *libunwind*, *pyelftools*, and *addr2line*.

libunwind is a portable and efficient C library that is very helpful in unwinding a stack (i.e., the process of removing function entries from function call stack at run time) from within a running program [32]. Although this library is very efficient in getting the call stack information, it is limited because it can only get the function and the register but cannot get the line number. To achieve this goal, debug information is needed to connect the performance issues and their true root causes in the source code.

[§]https://man7.org/linux/man-pages/man3/backtrace.3.html

```
1 0x6c986b, /h5bench/e3sm/src/drivers/e3sm_io_driver_h5blob.cpp:226
2 0x457c4b, /h5bench/e3sm/src/cases/var_wr_case.cpp:448
3 0x454e87, /h5bench/e3sm/src/cases/e3sm_io_case.cpp:99
4 0x452947, /h5bench/e3sm/src/e3sm_io_core.cpp:97
5 0x451f1c, /h5bench/e3sm/src/e3sm_io.c:563
6 0x4525da, /home/abuild/rpmbuild/BUILD/glibc-2.31/csu/../sysdeps/x86_64/start.S:122
```

Fig. 5. Mapping of stack addresses to source-code lines.

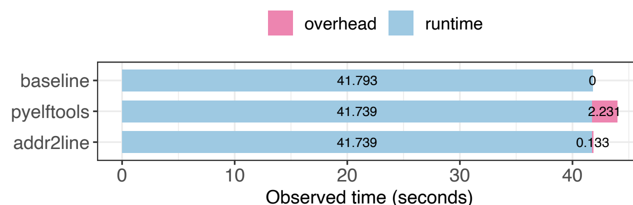


Fig. 6. Overhead for getting the line number from the addresses returned by `backtrace()` using different libraries.

To get the line number from the addresses, one needs access to DWARF (Debugging With Attributed Record Formats) [33], used by many compilers and debuggers to support source-level debugging [34]. *pyelftools* is a Python library for parsing and analyzing DWARF debugging information. This library provides the functionality to take in an address and the binary and return the function name, file name, and line number. Apart from *pyelftools*, there is a command line utility called *addr2line*, which uses the debugging information to translate addresses into file names and line numbers [35]. This utility takes as input a hexadecimal address and the binary and returns the file name and the line number associated with that address. A sample of this output is available in Fig. 5, which shows the mapping of the addresses reported for the `h5bench_e3sm` binary (in purple) in Fig. 4.

1) Feasibility: We prototyped a simple solution to compare *addr2line* and *pyelftools* on the h5bench [36] write benchmark and AMReX I/O kernel. We modified h5bench to collect the call stack using the `backtrace()` function call during the benchmark execution, where calls to writing the dataset are issued, and saved the returned addresses in a shared file. Once the stack addresses were available, we used both *pyelftools* and *addr2line* to get the line numbers in two separate implementations. These took as input the addresses in the file, retrieved the line information, file name, and function name, and reported the time it took to complete this step. For the h5bench write benchmark, we observed that the *pyelftools* library took considerably more time to get the line information as compared to *addr2line*, as can be shown in Fig. 6.

We noticed a similar trend with the AMReX kernel. Since there was a huge difference in overhead for getting line numbers between *pyelftools* and *addr2line*, we investigated *pyelftools* further, breaking down the time to get the line numbers and function names. The results in Fig. 7 show that getting the function names atone for most of this overhead. Since our experiments with the initial prototypes on the h5bench showed better performance and less overhead with *addr2line*, and though having the function name can be considered a plus when reporting this information back to the user, we found it sufficient to have the filename and line numbers only. They

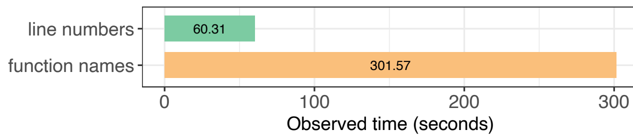


Fig. 7. Time taken to get the line numbers vs. the function names using *pyelftools* for all addresses returned by `backtrace()` in an execution with the AMReX [37] I/O kernel (1 compute node and 8 ranks).

would still reach the goal of pinpointing where in the code that I/O behavior originated from without the extra overhead. Hence, we opted to rely on *addr2line* to collect this data.

2) Enhancing Darshan DXT: To support getting the call-chain information in Darshan, we enhanced some data structures used by Darshan runtime to store the required information. These changes are depicted in Fig. 8. We modify the segment structure of the POSIX and MPIIO DXT records to store the call stack addresses in a buffer, apart from storing the usual information: the offset, transfer size, start, and end time. For each POSIX and MPI-IO read/write request, we retrieve the call stack of that request using `backtrace()`.

Apart from storing the call stack information for each read/write request, we also filter the unique address-to-line mappings for the binary while it is being instrumented by Darshan and store those in the Darshan log for later use. Through this approach, we avoid the dependency on the binary being always available to get the line information and allow log analysis with this additional information across multiple systems. To achieve this, we modified the POSIX and MPI-IO serialize functions, which are called by Darshan's shutdown routine to serialize all the records, and append them to the final Darshan log. In the serialize functions, we use `backtrace_symbols()` to get the symbolic representation of all the unique stack addresses collected.

Fig. 4 shows the output of `backtrace_symbols()` for the call stack addresses of a POSIX write call accessed in the POSIX serialize method. Multiple addresses are in the backtrace, but not all correspond to the binary. Some are from Darshan and HDF5 external libraries, which are not of interest, and processing these addresses will only add extra overhead. Through this symbolic representation, we check which addresses correspond to the binary. We store these addresses in a file-per-process approach to avoid extra communication and synchronization costs. In the next step, as part of the Darshan shutdown routine, we filter the unique addresses across these files and call *addr2line* to get the address-to-line mappings while we still have access to the binary. Fig. 5 shows the output of *addr2line* on some of the addresses that correspond to the application's binary. Through this technique, we can avoid calling *addr2line* on addresses that do not correspond to the binary, reducing the overall overhead. Once we have all the unique address-to-line mappings, we append these mappings to the header of the Darshan log.

To expose this new information appended in the Darshan log so external applications such as *Drishti* can benefit from it, we

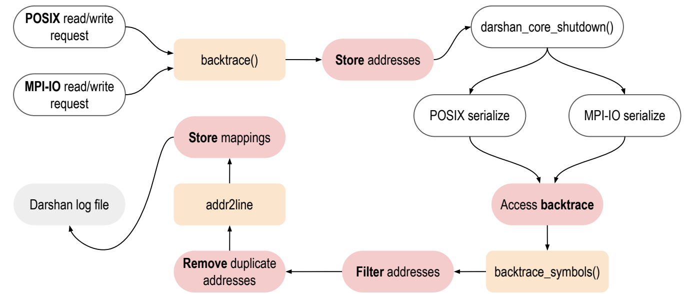


Fig. 8. Framework to capture source code information in Darshan to enhance I/O analysis and recommendations in *Drishti*.

modified both its utility package and PyDarshan. PyDarshan extends Darshan's analysis capabilities with a convenient Python interface and corresponding CLI utilities to enable advanced and customized analysis seamlessly connected to the rich ecosystem of data science and machine learning libraries that support Python. Since *Drishti* relies on PyDarshan to parse a Darshan log, we enhanced the pandas dataframe used to represent each access to include the stack memory addresses as a new column. We also added two more dataframes, one for POSIX and one for MPIIO, for the unique address-to-line mappings, using the addresses as unique identifiers. Fig. 8 illustrates the complete framework.

So far, we have discussed the framework developed to collect the stack trace information for any parallel application being instrumented by Darshan. To pinpoint the exact line number where the user needs to make optimizations, we combine the DXT information and the stack memory addresses to give the complete backtrace of the issue. For example, one of the issues that we look for in the HPC I/O applications is a high number of small read or write requests on the POSIX level. We formulate a threshold value below which any read or write request is considered small. Using the DXT information, we check which read or write requests have a size smaller than the threshold. Once we identify all the ranks that exhibited this behavior, we group those together and then provide the backtrace information to drill down to where the read/write call originated. We have implemented over 30 triggers in *Drishti*, of which 13 can be related to the application's source code rather than a misconfiguration. Those cover triggers include POSIX read/write count intensiveness, POSIX random read/write usage, POSIX size imbalance, MPIIO blocking read/write usage, etc. These triggers were enhanced to drill down and point to the origins of an I/O performance bottleneck in the source code.

3) Discussion and Overhead: One of the most significant advantages of this source code analysis framework is that it does not rely on the availability of the binary or code on the *Drishti* side, as we already get all the address-to-line mappings during instrumentation. This approach promotes flexibility and portability in analyzing the HPC I/O application. *Drishti* can get the address-to-line mappings offline as well but this will make the analysis more stringent as we are dependent on the availability of the binary. However, collecting the mappings

during instrumentation comes with a non-negligible cost, as it adds some overhead in Darshan to call *addr2line*, which requires invoking another process. We rely on `posix_spawn()`[§] instead of using the standard `system()` call to optimize this further. We noticed that `posix_spawn()` took less time to invoke the *addr2line* command to get the line numbers than the alternative.

Apart from this, we have also made sure that we call *addr2line* the least amount of times possible. To do that, we used the `backtrace_symbols()` to extract only the addresses that correspond to the binary, as discussed in Section III-A2. Nonetheless, we still observe some overhead in runtime. We noticed an increase in time for the E3SM application compared to when both Darshan and DXT trace collection are enabled. However, it is important to recall that such I/O debugging and tuning exercises often use small-scale experiments and then are scaled back to production-size runs. Hence, we consider this overhead acceptable for our potential gains based on the insights and recommendations provided by *Drishti*, as showcased in Section V.

Debug symbols are necessary to identify the root cause of performance issues in the code. However, despite their availability *Drishti* can still provide valuable insights using basic profiling metrics collected by Darshan in production systems. The initial report can be further enhanced by enabling the full range of features, including stack collection. One can draw a parallel with the same tradeoffs of overhead/value a debugger tool would bring to the table, with a focus on I/O access patterns and taking a step further by providing actionable insights.

Finally, we also make this stack trace collection in Darshan configurable through an environment variable, which is turned off by default, so the user can avoid incurring the additional overhead if they do not want this analysis. We will discuss the overhead further in the use cases section.

IV. Building Up to High Level Libraries

As discussed in Section II-D, both Darshan and Recorder have some limitations while collecting metrics and traces in high-level libraries, particularly HDF5 which it partially covered in both. Towards solving these issues, and seeking to balance usability and performance to demonstrate how *Drishti* could harness high-level metrics, we propose a VOL (Virtual Object Layer) connector [38] that HDF5-based applications can use without source-code modifications. Nonetheless, *Drishti* can be extended to handle other sources of data collected from high-level libraries. This paper uses HDF5 as a use case due to its inherent support for intercepting I/O calls using the VOL feature. The VOL is an abstraction layer in the HDF5 library that intercepts all API calls that could potentially access objects in an HDF5 container and forwards those calls to a VOL connector, which implements the storage. The user or application benefits from the familiar and widely-used HDF5 data model and public API. However, not all public HDF5 API calls pass through the VOL, only those that manipulate the storage. Since we are interested in the datasets and attribute operations, we are not bound by this limitation.

VOL connectors can be implemented as passthrough or terminal connectors. The first performs operations and then invokes another connector layer underneath, whereas the latter do not and are typically designed to map HDF5 objects and metadata to storage. The proposed *Drishti* VOL connector, by nature, fits the description of a passthrough VOL since we only want to capture operation's timestamps and meta-information for further analysis using *Drishti*.

An HDF5 file is a container for data and metadata. Regarding data operations, we focus on the HDF5 datasets. A dataset object eventually find its way to a file and it is stored in two parts: a header and a data array (i.e., the raw data represented as a one-dimensional or multi-dimensional array of elements). The `H5D` API provides mechanisms for managing datasets, including transferring data between memory and disk.

`H5Dcreate`, used to create an HDF5 dataset, could result in I/O operations if file space allocation is set. However, HDF5 exposes the `H5Pset_fill_value()` API which is designed to work in concert with both `H5Pset_alloc_time()` and `H5Pset_fill_time()`. The last two govern the timing of dataset storage allocation and fill value write operations and are important in tuning I/O performance. `H5Dcreate` can also result in small metadata writes, if metadata cache is not enabled, or HDF5 decides to flush it at the time. However, that metadata would be considered internal to the library.

Regarding metadata, HDF5 files can contain: (1) library metadata, (2) static user metadata, or (3) dynamic user metadata. Users do not have any direct interaction with or control over library metadata, as the HDF5 library itself generates that to describe the file structure and the contents of objects in a file. However, that is not the case for static and dynamic user metadata, which are defined and provided by the user. Examples of static metadata include property lists, dataset's dataype and dataspace, fill values, and dataset or group storage properties. It is also uncommon for this type of metadata to change through the life of a file or object. On the other hand, dynamic user metadata can change over time, and it is often stored in an HDF5 attribute.

Considering the aspects upon which a user might have control in tuning or optimizing, only the static and dynamic metadata would be of interest. Yet, when considering VOL connectors, non-storage HDF5 API calls do not go through the VOL (e.g., dataspace and property list calls); hence, we focus mainly on the dynamic user metadata defined by the HDF5 attributes API. An attribute has two parts: name and value(s). Attributes are assumed to be very small and are managed through a special interface, `H5A`, designed to attach attributes to primary data objects as small datasets containing metadata information and to minimize storage requirements.

Considering the key attributes operations (`H5Acreate`, `H5Aopen`, `H5Awrite`, `H5Aread`, and `H5Aclose`) and how they would eventually translate to file operations in underlying layers of the I/O stack, our tracing VOL connector should track

[§]https://man7.org/linux/man-pages/man3/posix_spawn.3.html

TABLE I
HDF5 DATASET AND ATTRIBUTE API OPTIONS CURRENTLY SUPPORTED BY THE *Drishti* I/O TRACING VOL CONNECTOR.

	Operation	File Operations	*Drishti*-VOL
Datasets	H5Dcreate	✓	✗
	H5Dopen	✗	✗
	H5Dwrite	✓	✓
	H5Dread	✓	✓
	H5Dclose	✗	✗
Attributes	H5Acreate	✗	✗
	H5Aopen	✗	✗
	H5Awrite	✓	✓
	H5Aread	✓	✓
	H5Aclose	✗	✗

both write and read operations only. It is important to notice that H5Acreate creates the attribute in memory, which only exists in the file once H5Awrite is called, while H5Aread generally comes into play in concert with H5Aget_* and H5Aopen_* functions to read from a file. Table I summarizes the support and tracing coverage of dataset and attribute operations in our VOL connector.

There are some caveats to ensure such VOL traces can be combined and matched with Darshan DXT traces. First, the DXT module stores the timestamp of each MPI-IO and POSIX operations relative to the start of the execution instead of relying on the absolute timestamp. To ensure data is collected through this tracing VOL connector, we adopted the same approach to collect the timestamp as Darshan uses. Still, we also require an offline adjustment to the relative format using the job start time reported by Darshan (which might differ from the actual job start time in milliseconds due to the initialization of Darshan itself). Second, to avoid additional overhead, the VOL traces are stored in memory and persisted to file using a file-per-process approach once the VOL is shut down. We opted for such a file-per-process approach to avoid adding message communication and potentially impacting the observed makespan time of the application. Lastly, since we are generating those additional files, Darshan will capture metrics regarding those operations. Nonetheless, we can easily filter them out when analyzing, visualizing, and recommending actions to avoid common I/O performance pitfalls.

The proposed VOL connector wraps the operations of interest (Table I) with microsecond-precision timers. For each operation, to ensure we could combine the reported metrics and give similar context information, we record the start, end, duration, rank, operation, and offset (where applicable).

V. CROSS-LAYER EXPLORATION

In this section, we demonstrate the potential of our cross-layer *Drishti* exploration solution with three use cases from distinct science domains, by harnessing different I/O metrics.

A. WarpX

WarpX [39] is a time-based, electromagnetic, and electrostatic Particle-In-Cell code that is highly parallel and optimized for GPUs and multi-core CPUs. WarpX scaled to the world's largest supercomputers and was awarded the 2022 ACM Gordon Bell Prize. It writes diagnostics data in plotfile or openPMD [40] format(s). While plotfiles are AMReX's native data format, openPMD is implemented in popular community formats such as ADIOS [41] and HDF5 [42]. Thus, the optimization of data access using HDF5 plays a crucial role in improving the application's data access performance.

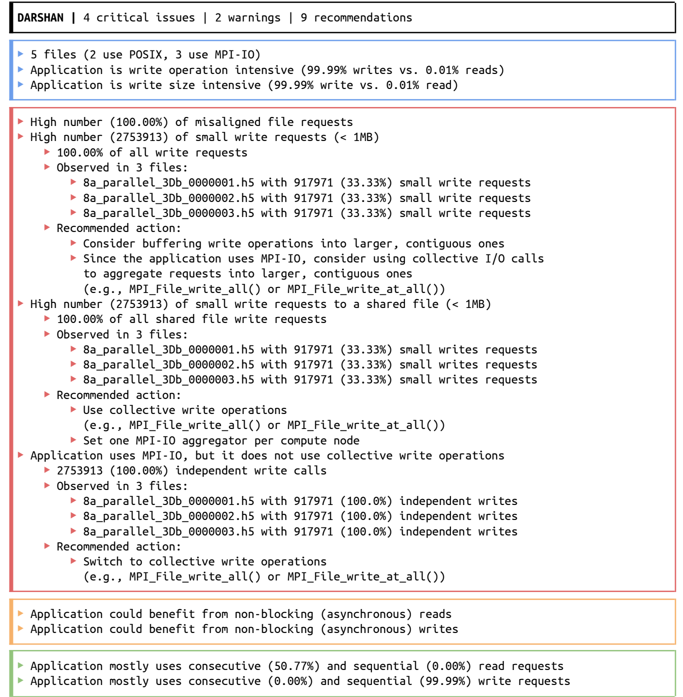
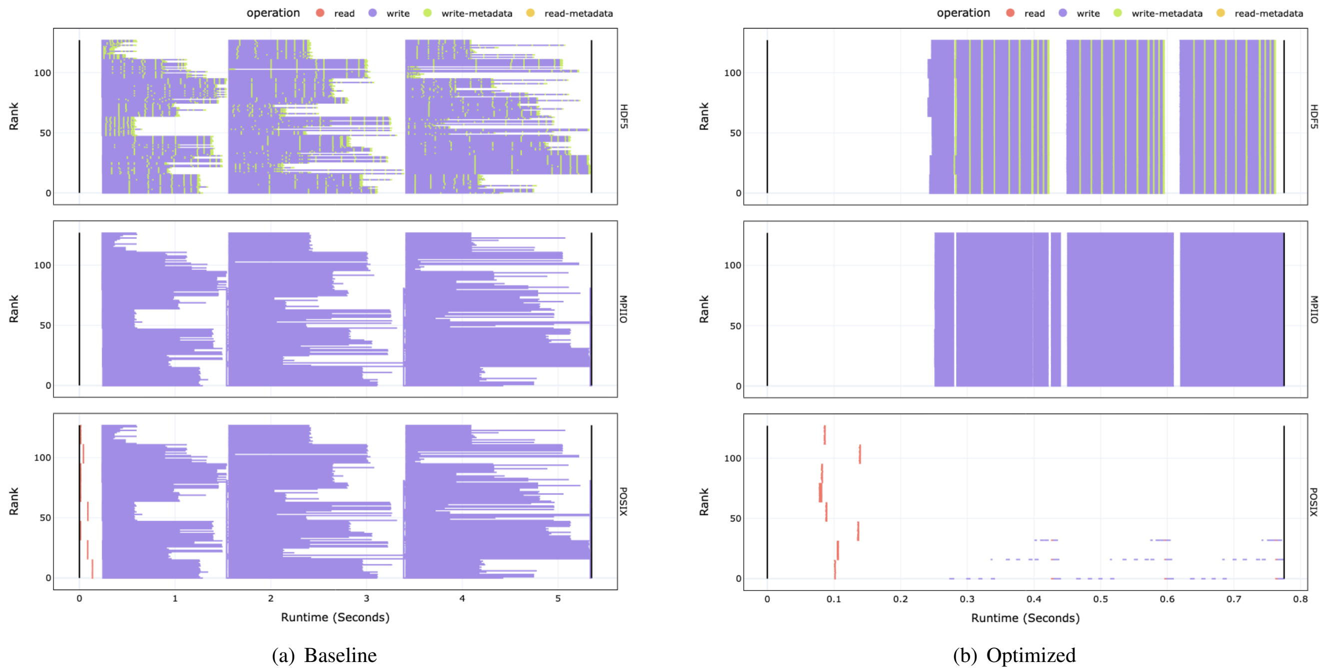

```
DARSHAN | 4 critical issues | 2 warnings | 9 recommendations

▶ 5 files (2 use POSIX, 3 use MPI-IO)
▶ Application is write operation intensive (99.99% writes vs. 0.01% reads)
▶ Application is write size intensive (99.99% write vs. 0.01% read)

▶ High number (100.00%) of misaligned file requests
▶ High number (2753913) of small write requests (< 1MB)
  ▶ 100.00% of all write requests
  ▶ Observed in 3 files:
    ▶ 8a_parallel_3Db_0000001.h5 with 917971 (33.33%) small write requests
    ▶ 8a_parallel_3Db_0000002.h5 with 917971 (33.33%) small write requests
    ▶ 8a_parallel_3Db_0000003.h5 with 917971 (33.33%) small write requests
  ▶ Recommended action:
    ▶ Consider buffering write operations into larger, contiguous ones
    ▶ Since the application uses MPI-IO, consider using collective I/O calls
      to aggregate requests into larger, contiguous ones
      (e.g., MPI_File_write_all() or MPI_File_write_at_all())
▶ High number (2753913) of small write requests to a shared file (< 1MB)
  ▶ 100.00% of all shared file write requests
  ▶ Observed in 3 files:
    ▶ 8a_parallel_3Db_0000001.h5 with 917971 (33.33%) small writes requests
    ▶ 8a_parallel_3Db_0000002.h5 with 917971 (33.33%) small writes requests
    ▶ 8a_parallel_3Db_0000003.h5 with 917971 (33.33%) small writes requests
  ▶ Recommended action:
    ▶ Use collective write operations
      (e.g., MPI_File_write_all() or MPI_File_write_at_all())
    ▶ Set one MPI-IO aggregator per compute node
▶ Application uses MPI-IO, but it does not use collective write operations
  ▶ 2753913 (100.00%) independent write calls
  ▶ Observed in 3 files:
    ▶ 8a_parallel_3Db_0000001.h5 with 917971 (100.0%) independent writes
    ▶ 8a_parallel_3Db_0000002.h5 with 917971 (100.0%) independent writes
    ▶ 8a_parallel_3Db_0000003.h5 with 917971 (100.0%) independent writes
  ▶ Recommended action:
    ▶ Switch to collective write operations
      (e.g., MPI_File_write_all() or MPI_File_write_at_all())

▶ Application could benefit from non-blocking (asynchronous) reads
▶ Application could benefit from non-blocking (asynchronous) writes

▶ Application mostly uses consecutive (50.77%) and sequential (0.00%) read requests
▶ Application mostly uses consecutive (0.00%) and sequential (99.99%) write requests
```

Fig. 9. Cross-layer I/O report and insights for openPMD extracted from *Drishti* VOL connector combined with Darshan metrics and DXT traces.

In this experiment, we evaluate openPMD's WarpX backend using HDF5 files, in a smaller debug-like scale to pinpoint the root causes of I/O performance problems. We used 8 compute nodes (maximum allowed number of nodes in Perlmutter's debug queue), 16 ranks per node, and a total of 128 processes. We used the PrgEnv-gnu/8.3.3 and cray-mpich/8.1.25. All dependencies were compiled with gcc/11.2.0, including the HDF5 library version 1.14.0. By default, one file is generated after each step. The total file size generated by each step is of $\approx$ 41MB for this small setup, with no compression set at the HDF5 level. We configured the kernel to write a few meshes and particles in 3D. The meshes are viewed as a grid of dimensions $[16 \times 8 \times 8]$ of mini blocks whose dimensions are $[16 \times 8 \times 4]$. Thus, the actual mesh size is $[256 \times 64 \times 32]$. For debugging purposes, since the application's I/O behavior does not change in between iterations, we set the kernel's execution to halt after writing three checkpoints.

Fig. 10 illustrates our investigation. The baseline (Fig. 10(a)) represents the default I/O behavior of the application before applying any optimization. First, if we consider only the two facets captured from the Darshan DXT module (i.e., MPIIO and POSIX), one can see that they look almost the same. Since the application uses MPI-IO, we can safely assume that no changes are being done at this layer, especially related



(a) Baseline (b) Optimized

Fig. 10. Interactive web-based cross-layer visualization of I/O requests for a execution of WarpX (OpenPMD) using multi-source metrics (*Drishti* VOL connector traces, DXT, and Darshan). Optimized execution time has a speedup of $6.9\times$ compared to the baseline.

to collective I/O calls, which would result in transformations at the POSIX level. Second, since accesses are independent, we can observe a time imbalance between the ranks. We can confirm that the workload between ranks is the same. Third, by including the traces from the *Drishti* VOL connector, we have a complete view from the application to lower levels of the stack. This addition is particularly interesting for openPMD since it uses a lot of dynamic user-level HDF5 metadata. Furthermore, metadata access occurs independently multiple times during each step. The I/O insights generated by *Drishti* are detailed in Fig. 9. Based on the metrics and traces *Drishti* has identified that the application only issues misaligned small requests to the file system. Here, we consider a request to be small if it is less than the Lustre stripe size used by the system (i.e., 1 MB). All accesses to these files suffer from the same problem. Furthermore, *Drishti* can capture the intended access of using shared files, and because the I/O operations were not issued collectively, it reports the high use of independent calls, suggesting the source code change.

From those triggered issues, we followed the recommendations of (1) aligning the I/O requests to the file system's stripe boundaries; (2) enabling collective I/O for data operations; (3) enabling collective I/O for HDF5 metadata operations. Figure 10(b) shows the optimized behavior of the application, and the impact of the changes. Total runtime was reduced from 5.351 seconds to 0.776 seconds, a $6.9\times$ speedup.

We also investigate the non-negligible overhead of collecting data from multiple layers of the I/O stack. Table II summarizes the results of five repetitions of each experiment by comparing the baseline execution with adding each layer of extra monitoring in terms of added runtime and total size of metrics. As expected, when enabling tracing the overhead

TABLE II
METRIC COLLECTION OVERHEAD FOR THE CROSS-LAYER ANALYSIS.

	Runtime (seconds)			Overhead	Combined
	Min.	**Median**	**Max.**	**(Min. %)**	**Log/Trace**
Baseline	5.99	7.52	8.62	-	-
+ Darshan	6.59	8.03	8.57	+9.62	35.88 KB
+ DXT	6.76	7.53	8.51	+3.03	38.88 MB
+ VOL	7.09	8.73	11.19	+4.88	41.69 MB

is increased since every I/O call is intercepted by Darshan. Furthermore, as demonstrated by Fig. Fig. 9 and Fig. 10(a), openPMD issues a lot of small requests, making the application more sensible to the tracing overhead (i.e., more small operations to trace rather than a few larger operations). The total file size of collect metrics also reflects the impact of tracing every I/O call at MPI-IO and POSIX layers, especially if independent operations are there which would imply in quite similar traces for both layers if compared to the aggregated ones found when collective I/O calls are used. These highlight that the impact of the overhead, when traces are collected, has a relation to how good or bad and application access its data. As for the VOL collected-metrics, we can see an added overhead of $\approx 5\%$ in time and 2.81 MB in size. We consider this acceptable in exchange for the gains attained from *Drishti*'s recommendations.

B. AMReX

AMReX [43] is a framework for massively parallel, block-structured adaptive mesh refinement (AMR) applications to solve partial differential equations on block-structured meshes. It is used in astrophysics, atmospheric modeling, combustion, cosmology, multi-phase flow, and particle accelerators.

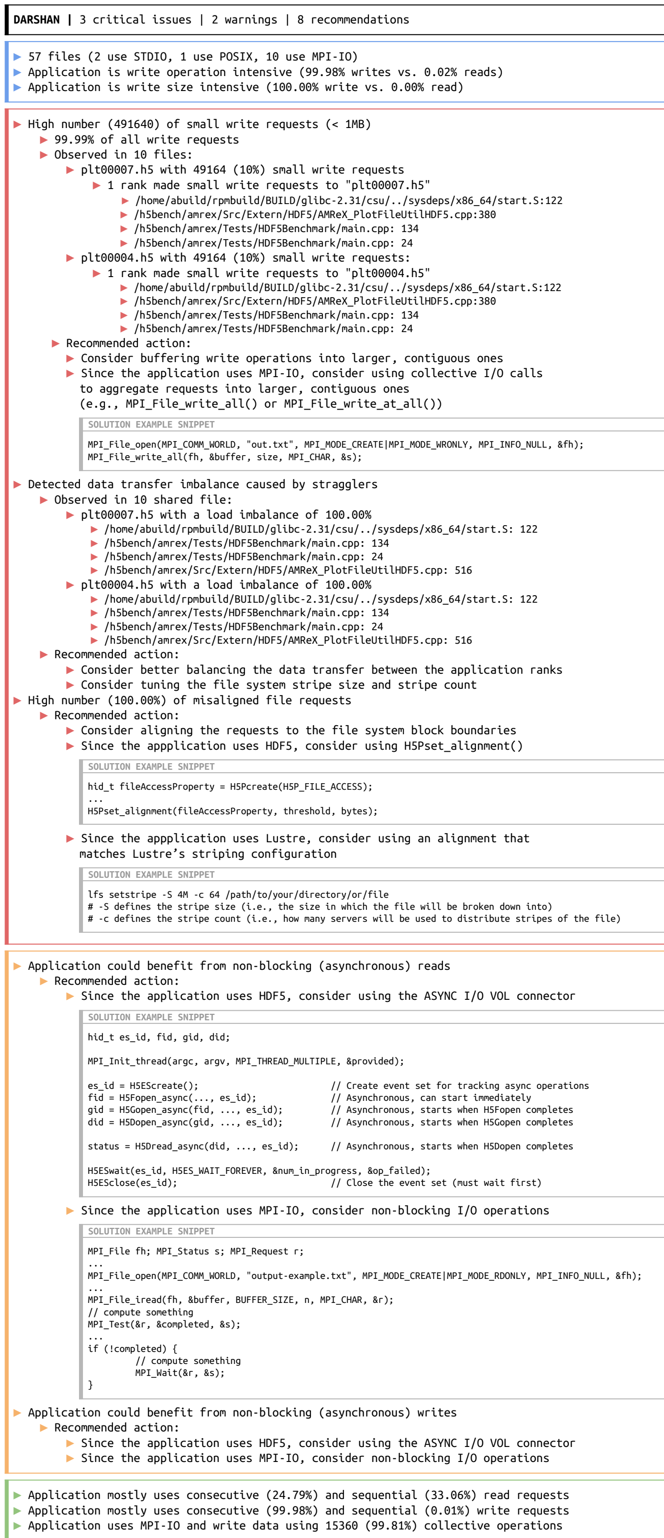


Fig. 11. Cross-layer I/O report and insights for the baseline (run-as-is) execution of AMReX in Perlmutter (NERSC) based on Darshan metrics/traces. This sample was generated with the verbose mode which includes source-code and configuration snippets.

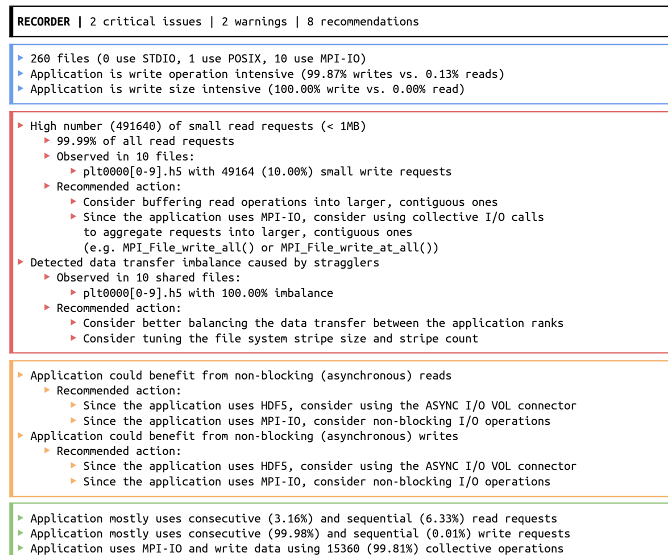


Fig. 12. Cross-layer I/O report and insights for the baseline (run-as-is) execution of AMReX in Perlmutter (NERSC) based on Recorder metrics/traces.

We ran AMReX with 512 ranks over 32 nodes in Perlmutter supercomputer, with a 1024 domain size, a maximum allowable size of each subdomain used for parallel decomposal as 8, 1 level, 6 components, 2 particles per cell, 10 plot files, and a sleep time of 10 seconds between writes. We collected I/O metrics and traces with Darshan, DXT, and the stack analysis enabled, and for comparison, with Recorder.

The cross-layer I/O report generated by *Drishti* for the baseline execution of AMReX using Darshan is shown in Fig. 11. The report indicates that the majority of write requests are small ($<$ 1MB) for all 10 plot files, and by harnessing our backtrace-based solution, it drills down to the source code by showing the line numbers and the function name from where the call originated from. We show the backtrace for 2 out of 10 files only in the report for brevity, but this can be changed as needed. To increase the size of these small write requests, we have set the stripe size to 16MB. Similarly, when accessing shared files, *Drishti* detects data transfer imbalance.

Fig. 12 shows the same report generated with data collected by Recorder. We omit the interactive visualization due to space constraints since those look relatively similar at first sight. However, as one might expect, there are some differences in the level of detail and recommendations that can be provided (discussed in Section II), aside from the source-code detection. First, Recorder reports a much larger number of files than Darshan, which is explained by the fact that Recorder intercepts all file accesses, (e.g., we detected 248 `/dev/shm/cray-shared-mem-coll-kvs*.tmp` files). That significantly skews two other metrics: the intensiveness ratio and the consecutive/sequential access ratio. Third, Recorder is unable to capture misaligned requests. This could be reconstructed based on the offset and size of each operation and the striping size for each file, but Recorder does not provide that.

Based on the recommendations from the cross-layer I/O report provided by *Drishti* with the source code information, we achieve a total speedup of 2.1× (from 211 to 100 seconds).

TABLE III
METRIC COLLECTION OVERHEAD FOR THE SOURCE CODE ANALYSIS.

	Runtime (seconds)			Overhead
	Min.	Median	Max.	(Min. %)
Baseline	4.60	4.85	5.97	-
+ Darshan	5.60	5.91	9.10	+21.68
+ DXT	7.00	8.87	8.87	+24.96
+ Stack	9.10	9.86	10.76	+30.03

C. Energy Exascale Earth System Model (E3SM)

E3SM-IO is the parallel I/O kernel from the Energy Exascale Earth System Model (E3SM) [44], [45] climate simulation model. It makes use of Parallel I/O Library (PIO) [46] which is built on top of PnetCDF [47]. For the evaluation in this paper, we used the F test case which is comprised of three unique data decomposition patterns shared by 388 2D and 3D variables (2 sharing Decomposition 1, 323 sharing Decomposition 2, and 63 sharing Decomposition 3).

The cross-layer I/O report generated by *Drishti* for the baseline execution of E3SM is shown in Fig. 13. The report highlights a high number of small read requests to one file. It also drill down exposing the file name and line numbers where those small requests originated. Similarly, *Drishti* also detects a high number of random read operations which constitute 37.89% of all read requests. This is observed in the same `.h5` file, but are triggered by another source-code region.

We also investigate the overhead for the source code analysis in this experiment. Table III summarizes these findings. As expected, there is an increase in time compared to the original run. The external system calls to *addr2line* mainly explain the overhead. We acknowledge that there is some overhead for performing this analysis, but that is only felt in the exploratory runs of the program when the user is trying

```
▸ High number (10878) of small read requests (< 1MB)
    ▸ 100% of all read requests
    ▸ Observed in 1 files:
        ▸ map_f_case_16p.h5 with 49164 (10%) small read requests
            ▸ 1 rank made small write requests to "map_f_case_16p.h5"
                ▸ /h5bench/e3sm/src/drivers/e3sm_io_driver.cpp: 120
                ▸ /h5bench/e3sm/src/drivers/e3sm_io_driver.cpp: 120
                ▸ /h5bench/e3sm/src/e3sm_io.c: 539 (discriminator 5)
                ▸ /home/abuild/rpmbuild/BUILD/glibc-2.31/csu/../sysdeps/x86_64/start.S: 122
    ▸ Recommended action:
        ▸ Consider buffering read operations into larger, contiguous ones
        ▸ Since the application uses MPI-IO, consider using collective I/O calls
          to aggregate requests into larger, contiguous ones
          (e.g., MPI_File_write_all() or MPI_File_write_at_all())
▸ High number (4122) of random read operations (< 1MB)
    ▸ 37.89% of all read requests
    ▸ Observed in 1 files:
        ▸ Below is the backtrace for these calls
            ▸ 1 rank made small write requests to "map_f_case_16p.h5"
                ▸ /home/abuild/rpmbuild/BUILD/glibc-2.31/csu/../sysdeps/x86_64/start.S: 122
                ▸ /h5bench/e3sm/src/cases/var_wr_case.cpp: 448
                ▸ /h5bench/e3sm/src/e3sm_io_core.cpp: 97
                ▸ /h5bench/e3sm/src/e3sm_io.c: 563
                ▸ /h5bench/e3sm/src/drivers/e3sm_io_driver_h5blob.cpp: 254
                ▸ /h5bench/e3sm/src/cases/e3sm_io_case.cpp: 136
    ▸ Recommended action:
        ▸ Consider changing your data model to have consecutive or sequential reads
▸  Application uses MPI-IO and issues 10877 (100.00%) independent read calls
    ▸ 10877 (100.0%) of independent reads in "map_f_case_16p.h5"
    ▸ Observed in 1 files:
        ▸ Below is the backtrace for these calls
                ▸ /h5bench/e3sm/src/e3sm_io.c: 539 (discriminator 5)
                ▸ /home/abuild/rpmbuild/BUILD/glibc-2.31/csu/../sysdeps/x86_64/start.S: 122
                ▸ /h5bench/e3sm/src/drivers/e3sm_io_driver_hdf5.cpp: 552
                ▸ /h5bench/e3sm/src/read_decomp.cpp: 253
    ▸ Recommended action:
        ▸ Consider using collective read operations and set one aggregator per compute node
          (e.g. MPI_File_read_all() or MPI_File_read_at_all())
```

Fig. 13. Critical issues detected by *Drishti* for the baseline (run-as-is) execution of E3SM in Perlmutter (NERSC) with Darshan metrics/traces.

to understand the I/O behavior of the application and tune it. Once the valuable insights are gained from *Drishti*, users will not have to incur additional overhead again. Moreover, we also noticed that the overhead remained the same and even reduced when the application scaled. Our experiments with 1024 ranks resulted in an increase of 11% in runtime compared to when both Darshan and DXT trace collection are enabled.

VI. RELATED WORK

Multiple profiling tools enable performance analysis and visualization of HPC I/O applications. These rely on different techniques to characterize and analyze I/O behavior and detect bottlenecks. We covered some in Section II, but we expand this discussion with other solutions and how *Drishti* can aid in closing some of the existing gaps.

TAU [16] is a popular portable profiling toolkit for instrumentation, measurement, and analysis of HPC applications. TAU can get CPU, memory, and communication metrics and extract serial and parallel file I/O information. It can also instrument external I/O libraries to characterize I/O performance using library wrapping. This allows TAU to intercept POSIX, MPI-IO calls, and instrument libraries such as HDF5. NVIDIA Nsight [48] is a commonly used tool to analyze and visualize the performance of HPC workloads. It provides meaningful insights such as the usage of CPU and GPU, vectorizations and parallelism, and GPU synchronization to optimize application performance. Other tools have a particular focus on I/O, such as IOMiner [49], Total Knowledge of I/O (TOKIO) [50], and Unified Monitoring and Metrics Interface (UMAMI) [51].

Regarding source code analysis, some tools employ this technique to detect bottlenecks in HPC applications. Drill [52] uses the source code of large-scale storage systems to train a sentiment language model, which is then used to detect how likely a runtime log entry is an anomaly. Drill also performs static code analysis on these storage systems to collect features through which it generates vector representations to train a bidirectional Long Short-Term Memory neural network.

HPCtoolkit [19] focuses on node-based performance analysis. It also uses source code analysis to get the program's structure from an application's binary. They build a mapping of machine instruction addresses in the binary to its context in the source code. HPCtoolkit relies on these mappings to attribute performance metrics to parts of source code, such as inlined functions and loops. This helps in further understanding the performance of the application.

Some work has also been done to connect the gap between I/O bottleneck detection and tuning. DXT Explorer [23] is one such tool that tries to fill this gap by proposing an interactive log-based web analysis tool to visualize the I/O behavior of the application from DXT logs. It provides interactive visualizations exploring different facets of the I/O behavior of the application with zoom-in and zoom-out capabilities. Drishti [24], [25] is another such tool that pinpoints the root causes of I/O performance pitfalls by evaluating the application based on certain triggers and categorizing the I/O

behavior based on the severity of the issue. It then provides a set of actionable recommendations to the user.

The tools mentioned above are effective in the performance analysis of HPC applications, some with a particular focus on I/O. However, due to limitations in the trace collection strategy and the ability to translate from the detected bottlenecks and optimizations, these tools preclude the end-user from seeing the complete picture related to the I/O performance of their application. *Drishti* solves this problem by providing a solution that uses cross-layer analysis, combining multiple performance metrics pertaining to the I/O software layers and drills all the way down to the source code level to identify the root causes of I/O bottlenecks, providing a visualization that helps understand the transformations the requests underwent until reaching lower levels of the HPC I/O stack.

VII. Conclusion

The complexity of the HPC I/O stack combined with gaps in the state-of-the-art profiling tools creates a barrier that does not help end-users and scientific application developers solve the I/O performance problems they encounter. Closing this gap requires cross-layer analysis combining multiple metrics and, when appropriate, drilling down to the source code. *Drishti* explores how various sources of I/O can be combined (while a standard is not in place) and harnessed to generate actionable insights for the end-user. It drills into the source code to provide valuable feedback on where the optimizations need to be made when those are not easily tuned by parameters. *Drishti* is open source and all of its components can be accessed at `https://github.com/hpc-io/drishti`.

Our results with *Drishti* show $2.1\times$ and $6.9\times$ speedup from run-as-is baseline executions on different applications. We also acknowledge the overhead that our implementation adds in Darshan; however, this overhead will not be present on all executions but only on exploratory small-scale runs often used to understand the I/O behavior, pinpoint I/O performance problems, and fine-tune the application. The user might have to incur some overhead on the initial execution of their application during such exploratory stages. However, once tuned according to the valuable insights provided by *Drishti*, they can scale up and benefit from the insights (without additional overheads) by turning off tracing.

Drishti currently relies on heuristics based on the HPC I/O community's collective experience to define its triggers. Creating a standard to represent I/O metrics and traces or even seamlessly being able to convert between divergent formats would greatly benefit the development of more advanced solutions that can further explore and extract insights from the complex interactions between layers of the I/O stack, catering to a much broader set of scientific applications.

Acknowledgment

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research was also supported by The Ohio State University under a subcontract (GR130303), which was supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research (ASCR) under contract number DE-AC02-05CH11231 with LBNL. This research used resources of the National Energy Research Scientific Computing Center under Contract No. DE-AC02-05CH11231.

References

[1] J. L. Bez, S. Byna, and S. Ibrahim, "I/O Access Patterns in HPC Applications: A 360-Degree Survey," *ACM Comput. Surv.*, vol. 56, no. 2, sep 2023. [Online]. Available: https://doi.org/10.1145/3611007

[2] M. Isakov, E. d. Rosario, S. Madireddy, P. Balaprakash, P. Carns, R. B. Ross, and M. A. Kinsy, "HPC I/O Throughput Bottleneck Analysis with Explainable Local Models," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[3] A. M. Karimi, A. K. Paul, and F. Wang, "I/O Performance Analysis of Machine Learning Workloads on Leadership Scale Supercomputer," *Performance Evaluation*, vol. 157-158, p. 102318, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166531622000268

[4] I. Peng, I. Karlin, M. Gokhale, K. Shoga, M. Legendre, and T. Gamblin, "A Holistic View of Memory Utilization on HPC Systems: Current and Future Trends," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '21. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3488423.3519336

[5] D. Reed, D. Gannon, and J. Dongarra, "HPC Forecast: Cloudy and Uncertain," *Commun. ACM*, vol. 66, no. 2, p. 82–90, jan 2023. [Online]. Available: https://doi.org/10.1145/3552309

[6] M. Vilayannur, R. Ross, T. Ludwig, J. Kunkel, S. Lang, and P. Carns, "Small-file Access in Parallel File Systems," in *2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2009, pp. 1–11.

[7] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 Characterization of Petascale I/O Workloads," in *2009 IEEE International Conference on Cluster Computing and Workshops (CLUSTER)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2009, pp. 1–10.

[8] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O Performance Challenges at Leadership Scale," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009.

[9] A. K. Paul, O. Faaland, A. Moody, E. Gonsiorowski, K. Mohror, and A. R. Butt, "Understanding HPC Application I/O Behavior Using System Level Statistics," in *IEEE 27th Int. Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2020, pp. 202–211.

[10] J. L. Bez, A. M. Karimi, A. K. Paul, B. Xie, S. Byna, P. Carns, S. Oral, F. Wang, and J. Hanley, "Access Patterns and Performance Behaviors of Multi-Layer Supercomputer I/O Subsystems under Production Load," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 43–55.

[11] A. R. Carneiro, J. L. Bez, C. Osthoff, L. M. Schnorr, and P. O. Navaux, "Uncovering I/O Demands on HPC Platforms: Peeking Under the Hood of Santos Dumont," *Journal of Parallel and Distributed Computing*, vol. 182, p. 104744, 2023.

[12] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation*, 1999, pp. 182–189.

[13] M. Agarwal, D. Singhvi, P. Malakar, and S. Byna, "Active Learning-based Automatic Tuning and Prediction of Parallel I/O Performance," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, 2019, pp. 20–29.

[14] B. Behzad, S. Byna, Prabhat, and M. Snir, "Optimizing I/O Performance of HPC Applications with Autotuning," *ACM Trans. Parallel Comput.*, vol. 5, no. 4, Mar. 2019.

[15] A. Bağbaba, "Improving Collective I/O Performance with Machine Learning Supported Auto-tuning," in *IEEE Int. Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 814–821.

[16] S. Shende, A. D. Malony, W. Spear, and K. Schuchardt, "Characterizing I/O Performance Using the TAU Performance System," in *Applications, Tools and Techniques on the Road to Exascale Computing, Proceedings of the conference ParCo 2011, 31 August - 3 September 2011, Ghent, Belgium*, ser. Advances in Parallel Computing, K. D. Bosschere, E. H.

D'Hollander, G. R. Joubert, D. A. Padua, F. J. Peters, and M. Sawyer, Eds., vol. 22. IOS Press, 2011, pp. 647–655.

[17] S. J. Kim, S. W. Son, W.-k. Liao, M. Kandemir, R. Thakur, and A. Choudhary, "IOPin: Runtime Profiling of Parallel I/O in HPC Systems," in *2012 SC: High Performance Computing, Networking Storage and Analysis*. IEEE Computer Society, 2012, pp. 18–23.

[18] A. Knüpfer and et al., "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir," in *Tools for High Performance Computing*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91.

[19] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs," *CCPE*, vol. 22, no. 6, pp. 685–701, 2010.

[20] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and Improving Computational Science Storage Access through Continuous Characterization," *ACM Trans. Storage*, vol. 7, no. 3, Oct. 2011.

[21] 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.

[22] 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.

[23] J. L. Bez, H. Tang, B. Xie, D. Williams-Young, R. Latham, R. Ross, S. Oral, and S. Byna, "I/O Bottleneck Detection and Tuning: Connecting the Dots using Interactive Log Analysis," in *2021 IEEE/ACM Sixth Int. Parallel Data Systems Workshop (PDSW)*, 2021, pp. 15–22.

[24] H. Ather, J. L. Bez, B. Norris, and S. Byna, "Illuminating The I/O Optimization Path Of Scientific Applications," in *High Performance Computing: 38th International Conference, ISC High Performance 2023, Hamburg, Germany, May 21–25, 2023, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2023, p. 22–41. [Online]. Available: https://doi.org/10.1007/978-3-031-32041-5_2

[25] 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)*, 2022, pp. 1–6.

[26] C. Xu, S. Snyder, O. Kulkarni, V. Venkatesan, P. Carns, S. Byna, R. Sisneros, and K. Chadalavada, "DXT: Darshan eXtended Tracing," *Cray User Group*, 1 2019.

[27] S. M. Inc, "High-Performance Storage Architecture and Scalable Cluster File System," Tech. Rep., 2007.

[28] A. George, R. Mohr, J. Simmons, and S. Oral, "Understanding Lustre Internals 2nd Edition," 9 2021. [Online]. Available: https://www.osti.gov/biblio/1824954

[29] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST '02. USA: USENIX Association, 2002, p. 19–es.

[30] P. Piela, "collectl Lustre Data Collection Plugins," 2021. [Online]. Available: https://github.com/pcpiela/collectl-lustre

[31] J. Garlick, "LMT3 - Lustre Monitoring Tool," 2010. [Online]. Available: https://github.com/LLNL/lmt

[32] "libunwind(3) — nongnu.org," https://www.nongnu.org/libunwind/man/libunwind(3).html, [Accessed 28-09-2023].

[33] D. D. I. F. Committee, "DWARF Debugging Information Format Version 5." [Online]. Available: https://dwarfstd.org/doc/DWARF5.pdf

[34] "IBM Developer — developer.ibm.com," https://developer.ibm.com/articles/au-dwarf-debug-format/, [Accessed 28-09-2023].

[35] "addr2line(1) - Linux manual page — man7.org," https://man7.org/linux/man-pages/man1/addr2line.1.html, [Accessed 28-09-2023].

[36] T. Li, S. Byna, Q. Koziol, H. Tang, J. L. Bez, and Q. Kang, "h5bench: HDF5 I/O Kernel Suite for Exercising HPC I/O Patterns," in *CUG*, 2021.

[37] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, M. P. Katz, A. Myers, T. Nguyen, A. Nonaka, M. Rosso, S. Williams, and M. Zingale, "AMReX: a framework for block-structured adaptive mesh refinement," *Journal of Open Source Software*, vol. 4, no. 37, p. 1370, May 2019.

[38] S. Byna, M. S. Breitenfeld, B. Dong, Q. Koziol, E. Pourmal, D. Robinson, J. Soumagne, H. Tang, V. Vishwanath, and R. Warren, "ExaHDF5: Delivering Efficient Parallel I/O on Exascale Computing Systems," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 145–160, Jan 2020.

[39] L. Fedeli, A. Huebl, F. Boillod-Cerneux, T. Clark, K. Gott, C. Hillairet, S. Jaure, A. Leblanc, R. Lehe, A. Myers, C. Piechurski, M. Sato, N. Zaim, W. Zhang, J.-L. Vay, and H. Vincenti, "Pushing the Frontier in the Design of Laser-Based Electron Accelerators with Groundbreaking Mesh-Refined Particle-In-Cell Simulations on Exascale-Class Supercomputers," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–12.

[40] Huebl, Axel and Lehe, Remi and Vay, Jean-Luc and Grote, David P. and Sbalzarini, Ivo F. and Kuschel, Stephan and Sagan, David and Mayes, Christopher and Perez, Frederic and Koller, Fabian and Bussmann, Michael. (2015) openPMD: A meta data standard for particle and mesh based data.

[41] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and Integration for Scientific Codes through the Adaptable IO System (ADIOS)," in *CLADE*. NY, USA: ACM, 2008, pp. 15–24.

[42] The HDF Group, "Hierarchical Data Format, version 5," 1997-. [Online]. Available: http://www.hdfgroup.org/HDF5

[43] W. Zhang, A. Myers, K. Gott, A. Almgren, and J. Bell, "AMReX: Block-structured adaptive mesh refinement for multiphysics applications," *The International Journal of High Performance Computing Applications*, vol. 35, no. 6, pp. 508–526, 2021.

[44] E3SM, "Energy Exascale Earth System Model v2.1.0," Jan 2023. [Online]. Available: https://doi.org/10.11578/E3SM/dc.20230110.5

[45] J.-C. Golaz, L. P. Van Roekel, X. Zheng, A. F. Roberts, J. D. Wolfe, W. Lin, A. M. Bradley, Q. Tang, M. E. Maltrud, R. M. Forsyth, C. Zhang, T. Zhou, K. Zhang, C. S. Zender, M. Wu, H. Wang, A. K. Turner, B. Singh, J. H. Richter, Y. Qin, M. R. Petersen, A. Mametjanov, P.-L. Ma, V. E. Larson, J. Krishna, N. D. Keen, N. Jeffery, E. C. Hunke, W. M. Hannah, O. Guba, B. M. Griffin, Y. Feng, D. Engwirda, A. V. Di Vittorio, C. Dang, L. M. Conlon, C.-C.-J. Chen, M. A. Brunke, G. Bisht, J. J. Benedict, X. S. Asay-Davis, Y. Zhang, M. Zhang, X. Zeng, S. Xie, P. J. Wolfram, T. Vo, M. Veneziani, T. K. Tesfa, S. Sreepathi, A. G. Salinger, J. E. J. Reeves Eyre, M. J. Prather, S. Mahajan, Q. Li, P. W. Jones, R. L. Jacob, G. W. Huebler, X. Huang, B. R. Hillman, B. E. Harrop, J. G. Foucar, Y. Fang, D. S. Comeau, P. M. Caldwell, T. Bartoletti, K. Balaguru, M. A. Taylor, R. B. McCoy, L. R. Leung, and D. C. Bader, "The DOE E3SM Model Version 2: Overview of the Physical Model and Initial Model Evaluation," *Journal of Advances in Modeling Earth Systems*, vol. 14, no. 12, p. e2022MS003156, 2022.

[46] E. Hartnett and J. Edwards, "The Parallel I/O (PIO) C/FORTRAN Libraries for Scalable HPC Performance," in *101st American Meteorological Society Annual Meeting (AMS)*, 01 2021.

[47] J. Li, W. keng 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 *ACM/IEEE Supercomputing*, ser. SC'03. Phoenix, AZ, USA: IEEE, 2003, pp. 39–39.

[48] NVIDIA, "Nsight Systems." [Online]. Available: https://developer.nvidia.com/nsight-systems

[49] T. Wang, S. Snyder, G. Lockwood, P. Carns, N. Wright, and S. Byna, "IOMiner: Large-Scale Analytics Framework for Gaining Knowledge from I/O Logs," in *CLUSTER*, 2018, pp. 466–476.

[50] 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 Cray User Group*, 1 2018.

[51] G. K. Lockwood, W. Yoo, S. Byna, N. J. Wright, S. Snyder, K. Harms, Z. Nault, and P. Carns, "UMAMI: A Recipe for Generating Meaningful Metrics through Holistic I/O Performance Analysis," in *2nd Joint Int. Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems*. New York, NY, USA: ACM, 2017, p. 55–60.

[52] D. Zhang, C. Egersdoerfer, T. Mahmud, M. Zheng, and D. Dai, "Drill: Log-based Anomaly Detection for Large-scale Storage Systems Using Source Code Analysis," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 189–199.