Towards End-to-End I/O Analysis and Optimization of HPC Systems

by

Hammad Bin Ather

A dissertation accepted and approved in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science

Dissertation Committee:
Hank Childs, Co-Chair
Allen D. Malony, Co-Chair
Jee Choi, Core Member
Kathryn Mohror, Core Member
Christopher H. Hendon, Institutional Representative

University of Oregon
Fall 2025

DISSERTATION ABSTRACT

Hammad Bin Ather

Doctor of Philosophy in Computer Science

Title: Towards End-to-End I/O Analysis and Optimization of HPC Systems

The I/O subsystem is a critical bottleneck in high-performance computing (HPC) systems, making its optimization essential for scalability and efficiency. However, the growing complexity of modern hardware and the diversity of scientific workloads have made I/O increasingly difficult to analyze and tune. Existing optimization efforts suffer from two key limitations: (1) a disconnect between I/O profiling/tracing data, the performance issues they indicate, and the actionable optimizations to address them; and (2) reliance on exhaustive parameter searches or costly machine learning models to optimize the I/O performance.

This dissertation addresses these limitations through two complementary, end-to-end approaches that bridge I/O analysis and optimization. First, it introduces a scalable and interactive I/O visualization and analysis approach that bridges the gap between trace collection, diagnosis, and tuning. This approach enables users to explore large-scale I/O traces, automatically detect bottlenecks, and generate targeted recommendations by combining interactive visualizations with cross-layer profiling and source-code insights. Second, it presents a lightweight runtime workflow that predicts recurring I/O patterns, extracts access insights, and applies on-the-fly optimizations using empirically derived rules. This approach eliminates the need for prior training, profiling, or parameter exploration, achieving dynamic I/O optimization with minimal overhead. Together, these approaches enable a continuous

cycle of analysis-informed optimization, advancing the state of end-to-end I/O characterization and performance improvement in HPC environments.

This dissertation includes previously published and unpublished coauthored material.

CURRICULUM VITAE

NAME OF AUTHOR: Hammad Bin Ather

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
FAST National University, Lahore, Pakistan

DEGREES AWARDED:

Master of Science, Computer Science, 2025, University of Oregon
Bachelor of Science, Computer Science, 2018, FAST National University

AREAS OF SPECIAL INTEREST:

High Performance Computing
Performance Analysis and Visualization
Parallel I/O
Distributed AI/ML Training

PROFESSIONAL EXPERIENCE:

Software Engineer Intern, Meta, June 2025 - September 2025

Computing Graduate Student Intern, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, June 2024 - September 2024

Student Assistant, Scientific Data Management Group, Lawrence Berkeley National Laboratory, June 2022 - December 2023

Graduate Employee Research, University of Oregon, Winter 2021, Fall 2021 - Spring 2023, Winter 2025 - Fall 2025

Graduate Employee Teaching, University of Oregon, Spring 2021, Fall 2023 - Fall 2024

Research Assistant, Lahore University of Management Sciences, February 2019 - August 2020

Software Engineer, i2c inc, June 2018 - February 2019

GRANTS, AWARDS AND HONORS:

Erwin and Gertrude Juilfs Scholarship for Research Excellence, University of Oregon, 2024

ACM SIGHPC Travel Award, 2024

HDF5 User Group Travel Award, 2023

Student Volunteer at Supercomputing 2022, Dallas, TX

PUBLICATIONS:

Hammad Bin Ather, Chen Wang, Hariharan Devarajan, Hank Childs, Kathryn Mohror. (2025). SmartIO: A Lightweight End-to-End Workflow for Runtime I/O Optimization of HPC Systems. *10th International Parallel Data Systems Workshop (PDSW)* held in conjunction with SC25

Hammad Bin Ather[1], Jean Luca Bez, Yankun Xia, Suren Byna. (2024). Drilling Down I/O Bottlenecks with Cross-layer I/O Profile Exploration. *38th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*

Hammad Bin Ather, Sophie Berkman, Giuseppe Cerati, Matti J. Kortelainen, Ka Hei Martin Kwok, Steven Lantz, Seyong Lee, Boyana Norris, Michael Reid, Allison Reinsvold Hall, Daniel Riley, Alexei Strelchenko, Cong Wang. (2024). Exploring code portability solutions for HEP with a particle tracking test code. *The Frontiers in Big Data, Sec. Big Data and AI in High Energy Physics*

Hammad Bin Ather, Jean Luca Bez, Boyana Norris, Suren Byna. (2023). Illuminating the I/O Optimization Path of Scientific Applications. *ISC High Performance*

Ka Hei Martin Kwok, Matti Kortelainen, Giuseppe Cerati, Alexei Strelchenko, Oliver Gutsche, Allison Reinsvold Hall, Steve Lantz, Michael Reid, Daniel Riley, Sophie Berkman, Seyong Lee, Hammad Bin Ather, Boyana Norris, Cong Wang. (2023). *26th International Conference on Computing in High Energy and Nuclear Physics (CHEP)*

[1]Joint first authorship with Jean Luca Bez.

Jean Luca Bez, Hammad Bin Ather, Suren Byna. (2022). Drishti: Guiding End-Users in the I/O Optimization Journey. *2022 IEEE/ACM International Parallel Data Systems Workshop (PDSW)* held in conjunction with SC22

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisors, Dr. Hank Childs and Dr. Allen D. Malony, for their guidance, encouragement, and mentorship throughout my Ph.D. I am especially thankful to Dr. Childs for his support both professionally and personally; I have found him not only an exceptional advisor, but also a genuinely kind and thoughtful person. I am also grateful to my former advisor, Dr. Boyana Norris, for introducing me to high-performance computing and supporting me during the early years of my doctoral studies.

I would like to acknowledge my mentors and collaborators at the national laboratories. At Lawrence Berkeley National Laboratory, I am thankful to Dr. Jean Luca Bez and Dr. Suren Byna for their support and mentorship. Jean and Suren are outstanding researchers in the field of HPC and parallel I/O, and it was a privilege to collaborate with them early in my Ph.D. At Lawrence Livermore National Laboratory, I am grateful to Dr. Kathryn Mohror, Dr. Chen Wang, and Dr. Hariharan Devarajan for their guidance and collaboration. Kathryn has generously supported my research for the past one and a half years and also served on my committee, for which I am deeply appreciative.

I would also like to thank Dr. Ihsan Ayyub Qazi, my research advisor at LUMS, for nurturing my enthusiasm for research and guiding me through the graduate school application process.

I would also like to thank the administrative staff of the Computer Science Department at the University of Oregon—particularly Nicole Moynahan and Cheri Smith—for their consistent support. I am also grateful to my colleagues in the CDUX

and HPCL groups for their camaraderie, encouragement, and helpful discussions throughout the program.

Finally, I am profoundly indebted to my family for their unwavering support. I thank my parents for encouraging my decision to pursue graduate studies far from home, and my siblings, Usama and Zarnaab, for believing in me throughout this journey. Most importantly, I am deeply grateful to my wife, Aliza Lisan, for her steadfast love, strength, and support—balancing her own Ph.D. while raising our child with me and standing by my side every step of the way. I would not have reached this milestone without her.

To my son, Nayel Hammad.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 19

1.1. Limitations in the State-of-the-Art 19

1.1.1. Limitation 1: Disconnect Between I/O Tracing and Optimization 19

1.1.2. Limitation 2: High Tuning Overhead 20

1.2. Thesis Topic 20

1.2.1. Bridging the Gap: An Interactive I/O Analysis Framework . . 21

1.2.1.1. Extracting I/O Behavior from Metrics 22

1.2.1.2. Exploring I/O Behavior Interactively 22

1.2.1.3. Automatic Detection of I/O Bottlenecks 23

1.2.2. Cross-Layer I/O Profiling and Source Code Analysis for Bottleneck Diagnosis 23

1.2.2.1. Cross-Layer I/O Profiling 23

1.2.2.2. Source Code Analysis 24

1.2.3. Runtime I/O Optimization Through a Lightweight End-to-End Workflow 24

1.2.3.1. Building the End-to-End Workflow 25

1.2.3.2. Major Components 25

II. STATE-OF-THE-ART IN PARALLEL I/O CHARACTERIZATION AND EVALUATION 27

2.1. Introduction 27

2.2. HPC I/O Stack 29

2.2.1. High-level I/O libraries 29

2.2.2. Parallel and low-level I/O libraries 31

2.2.3. I/O forwarding layer 32

Chapter Page

2.2.4. Parallel File System 32

2.2.5. Storage Hardware 34

2.3. Parallel I/O Characterization and Evaluation 34

2.3.1. Measurement and Statistics Collection 35

2.3.1.1. I/O Workloads 35

2.3.1.2. I/O Monitoring and Collection 37

2.3.2. Analysis, Modeling, and Prediction 49

2.3.2.1. Statistics and Analysis 49

2.3.2.2. Predictive Analysis 54

2.3.2.3. Replay-based Modeling 57

2.4. Conclusion 61

III. BRIDGING THE GAP: AN INTERACTIVE I/O ANALYSIS FRAMEWORK 62

3.1. Introduction 62

3.2. Visualization, Diagnosis, and Recommendations 65

3.2.1. Extracting I/O Behavior from Metrics 66

3.2.2. Exploring I/O Behavior Interactively 67

3.2.3. Automatic Detection of I/O Bottlenecks 70

3.2.4. Exploring I/O Phases and Bottlenecks 73

3.2.4.1. Blocking I/O Accesses 75

3.2.4.2. I/O Stragglers 76

3.2.5. Towards Exploring File System Usage 76

3.3. Evaluation 77

3.3.1. I/O Systems in NERSC and OLCF 78

3.3.2. I/O Bottlenecks in OpenPMD 78

3.3.3. Improving AMReX with Asynchronous I/O 80

Chapter Page

3.4. Conclusion . . . 82

IV. CROSS-LAYER I/O PROFILING AND SOURCE CODE ANALYSIS FOR BOTTLENECK DIAGNOSIS . . . 84

4.1. Introduction . . . 84

4.2. Data Sources and Limitation . . . 87

4.2.1. Darshan . . . 87

4.2.2. Darshan DXT . . . 87

4.2.3. Recorder . . . 88

4.2.4. High-Level Libraries . . . 89

4.2.5. Parallel File Systems (PFS) . . . 89

4.3. Drilling Down to the Source-Code . . . 90

4.3.1. Unveiling the Source-Code . . . 91

4.3.1.1. Feasibility . . . 94

4.3.1.2. Enhancing Darshan DXT . . . 95

4.3.1.3. Discussion and Overhead . . . 98

4.4. Building up to High Level Libraries . . . 100

4.5. Evaluation . . . 103

4.5.1. WarpX . . . 103

4.5.2. AMReX . . . 108

4.5.3. Energy Exascale Earth System Model (E3SM) . . . 111

4.6. Conclusion . . . 113

V. RUNTIME I/O OPTIMIZATION THROUGH A LIGHTWEIGHT END-TO-END WORKFLOW . . . 115

5.1. Introduction . . . 115

5.2. Architecture . . . 117

5.2.1. Building the End-to-End Workflow . . . 118

Chapter Page

5.2.1.1. Identifying the Start of an I/O Phase 119

5.2.1.2. Distinguishing Between New and Recurring I/O Phases 119

5.2.1.3. Applying Optimizations 120

5.2.2. Prediction Component 121

5.2.2.1. Pattern-Recognition 122

5.2.2.2. I/O Call Sequence Prediction 123

5.2.3. Extraction Component 125

5.2.4. Optimization Component 127

5.2.4.1. Evaluations for HDF5 Data Transfer Mode Rules 127

5.2.4.2. Evaluations for ROMIO Rules 128

5.2.4.3. Evaluations for Lustre Stripe Count Rules 129

5.3. Evaluation 132

5.3.1. IOR 132

5.3.2. Flash-X 135

5.3.3. Overhead 138

5.3.4. Comparison with the state-of-the-art 139

5.4. Limitations 141

5.5. Conclusion 141

VI. CONCLUSION AND FUTURE WORK 143

6.1. Conclusion 143

6.2. Future Work 145

6.2.1. Leveraging Large Language Models (LLMs) and Autonomous AI Agents 145

6.2.2. Expanding Library, Format, and System Coverage 147

Chapter Page

6.2.3. A Bold Future Direction: Toward Self-Optimizing and Self-Explaining HPC Systems 148

REFERENCES CITED 150

LIST OF FIGURES

Figure Page

1. Parallel I/O stack used in production-scale supercomputers 30
2. Taxonomy of different phases in parallel I/O evaluation 34
3. Overview of Darshan's architecture 40
4. Overhead of Darshan 40
5. Features of the Recorder framework 42
6. Overview of the IOPro framework 45
7. Maximum I/O throughput per application across three supercomputers . 53
8. Omnisc'IO architecture 56
9. Architecture of an autotuning framework 58
10. Workflow of a replay-based modeling framework 60
11. Major components of *Drishti* 65
12. Comparison of methods to extract the I/O behavior from traces 66
13. Comparison of solutions to generate the interactive plots 68
14. *Drishti* reports focusing on different facets of the I/O behavior 69
15. Interactive I/O phases visualization 75
16. Identifying stragglers through interactive visualization 76
17. Lustre data storage (OST) access over time 77
18. OpenPMD baseline execution analysis by *Drishti* 79
19. Darshan format to store I/O metrics from multiple modules/sources. . . 87
20. Recorder format representation to store I/O traces. 88
21. Sample backtrace from an HDF5 application monitored with Darshan . . 93
22. Mapping of stack addresses to source-code lines 94

Figure Page

23. Overhead for getting the line number from the addresses using different libraries 95

24. Time taken to get the line numbers vs. the function names using *pyelftools* 95

25. Source code analysis framework in *Drishti* 97

26. Interactive web-based cross-layer visualization of I/O requests for a WarpX (OpenPMD) execution 105

27. Cross-layer I/O report and insights for OpenPMD 106

28. Cross-layer I/O report and insights for the baseline execution of AMReX for Darshan metrics/traces 109

29. Cross-layer I/O report and insights for the baseline execution of AMReX for Recorder traces 110

30. E3SM baseline execution analyzed by *Drishti* 112

31. Overview of the *SmartIO* workflow 117

32. Example I/O code and the generated CFG and CST 122

33. I/O performance comparison of progressive and static striping 131

34. IOR bandwidth speedup with *SmartIO* 134

35. Reduction in IOR execution time with *SmartIO* 135

36. Flash-X bandwidth speedup with *SmartIO* 137

37. Reduction in Flash-X I/O time with *SmartIO* 138

38. Overhead analysis of *SmartIO* 139

39. Future direction of incorporating AI agents 146

LIST OF TABLES

Table Page

1. System-Level I/O Benchmark Comparison 36
2. Application-Level I/O Benchmark Comparison 36
3. I/O profiling, tracing, and monitoring tools and their characteristics . . . 44
4. Statistical I/O analysis tools and related research 50
5. Predictive I/O analysis tools 55
6. Replay-based I/O modeling tools and related research 59
7. Root causes of I/O performance bottlenecks 71
8. Triggers evaluated by *Drishti* for each Darshan log. 72
9. *Drishti* report generated for the AMReX in Cori 81
10. HDF5 dataset and attribute API options currently supported by the *Drishti* I/O tracing VOL connector. 102
11. Metric collection overhead for the cross-layer analysis. 107
12. Metric collection overhead for the source code analysis. 112
13. Function calls scanned in the predicted I/O call stack trace to extract I/O insights 125
14. Rules guiding the runtime optimization decisions in *SmartIO* 126
15. Effect of changing the default ROMIO collective buffering parameters on I/O performance 129
16. Mapping of insights to optimization rules for IOR optimization 133
17. Mapping of insights to optimization rules for Flash-X optimization . . . 137
18. Comparison of achieved speedup and tuning overhead on the IOR benchmark: *SmartIO* vs. state-of-the-art 140

CHAPTER I

INTRODUCTION

High-performance computing (HPC) is a critical technology for solving complex computational and scientific problems [6]. HPC systems run critical workloads with exceptional speed and efficiency, leveraging state-of-the-art hardware and computing resources. With their ability to run large-scale simulations and process massive volumes of data at unprecedented speeds, these systems set the foundation for driving innovation and scientific discovery.

Although current HPC systems are immensely powerful, harnessing their peak performance remains a significant challenge. These systems are composed of thousands of interconnected nodes, each comprising various system components, including computation, memory, I/O, and networking. For an HPC system to achieve optimal performance, these components must work together in harmony, and a bottleneck in any of them can reduce the overall performance. Among these components, the I/O subsystem is often a critical bottleneck [49], with various underlying causes, including unbalanced I/O workload [135], I/O resource contention [146], and other factors [25].

1.1 Limitations in the State-of-the-Art

Efforts have been made to optimize the I/O subsystem; however, they suffer from two key limitations:

1.1.1 Limitation 1: Disconnect Between I/O Tracing and Optimization. When applications experience I/O performance slowdowns, pinpointing the root causes of inefficiencies requires detailed metrics and a thorough understanding of the I/O stack. A variety of I/O performance profiling and tracing tools are available that collect I/O performance data for scientific applications. For

instance, Darshan [21], a commonly used I/O profiling tool, collects metrics to provide a coarse-grain view of the application's behavior when accessing data. Similarly, Recorder [131], an I/O tracing tool, can trace I/O requests and provide a fine-grain view of the transformations the requests undergo as they traverse the I/O software stack. However, all of these tools lack analysis and visualization capabilities, which hinders the end-users' ability to understand the issues the metrics and traces collected by these tools represent. Hence, despite the availability of such fine-grained traces, there is a gap between the trace collection, analysis, and tuning steps.

1.1.2 Limitation 2: High Tuning Overhead. Considerable work has been done to develop methodologies for optimizing parallel I/O in HPC applications; however, despite their effectiveness, these approaches often come with substantial time and resource overhead. For example, autotuning approaches [7, 8, 9] aim to identify optimal tuning parameters across different layers of the I/O stack, but they often incur significant overhead—sometimes requiring hours to converge on the best configuration. Machine learning-based I/O prediction and optimization [78, 125, 144] also faces overhead challenges, as it relies on extensive training data collected from multiple simulation runs, making it time and resource-intensive. Furthermore, machine learning based tools lack generality as they are designed for specific applications and libraries. Collectively, these approaches provide benefits in optimizing I/O performance but require a large overhead cost.

1.2 Thesis Topic

To address the aforementioned limitations, this dissertation answers two high-level research questions (RQs), each targeting a specific limitation. RQ1 corresponds to the first limitation, focusing on bridging the gap between trace data and actionable insights, while RQ2 addresses the second limitation, centered on reducing the

overhead of I/O performance tuning. Each high-level RQ is further supported by a set of sub-questions that guide the investigation.

RQ 1. How can the gap between low-level I/O traces, the performance issues they reflect, and the optimizations required to resolve them be bridged?

- **RQ 1.1** How can interactive visualization of I/O logs aid in diagnosing and understanding application-level I/O behavior?
- **RQ 1.2** How can cross-layer I/O profiling and static code analysis help identify the root causes of I/O bottlenecks?

RQ 2. How can the significant tuning overhead of state-of-the-art I/O optimization approaches be mitigated?

- **RQ 2.1** How can the I/O behavior be predicted and learned during the execution of the application?
- **RQ 2.2** How can runtime insights be translated into actionable optimizations on the fly with minimal overhead?

Co-author Acknowledgement Chapter II contains both unpublished and published material with co-authorship. Specific contributions are detailed in the beginning of the chapter, but co-authors include Dr. Jean Luca Bez, Dr. Chen Wang, Dr. Hank Childs, Dr. Allen D. Malony, and Dr. Suren Byna.

1.2.1 Bridging the Gap: An Interactive I/O Analysis Framework.

To address **RQ1**, this dissertation introduces an interactive, web-based I/O visualization and analysis approach, realized in an implementation named *Drishti*. *Drishti* is designed to: (1) allow users to interactively explore large-scale I/O traces, (2) provide contextual insights into how applications issue requests and how these

requests propagate across the I/O stack, and (3) automatically detect common I/O bottlenecks and recommend targeted optimizations. The overarching goal is to streamline I/O performance diagnosis, making bottleneck detection both scalable and accessible to experts and non-experts alike, while supporting a wide range of scientific and emerging AI/ML workloads. More details about *Drishti* are provided in Chapters III and IV

1.2.1.1 Extracting I/O Behavior from Metrics. *Drishti* leverages Darshan [21] and its tracing module DXT [143] to extract meaningful I/O behavior from application logs. Darshan [21] is widely deployed on large-scale HPC systems and collects profiling data with minimal overhead, while DXT [143] extends this functionality by capturing fine-grained POSIX and MPI-IO traces. Because DXT logs are stored in binary format, *Drishti* employs PyDarshan, a Python interface that parses these logs into *pandas* DataFrames, enabling scalable and efficient analysis of large application traces.

1.2.1.2 Exploring I/O Behavior Interactively. Analyzing I/O traces is challenging due to their size and the difficulty of spotting bottlenecks in static plots, which often obscure issues such as server-level interference across thousands of ranks. To address this, *Drishti* adopts interactive visualizations using the open-source *plotly* Python library, chosen for its compatibility with PyDarshan and ease of integration without adding cross-language dependencies. It generates per-file visualizations that expose I/O behavior from multiple perspectives (e.g., operations, data transfer, spatiality), with synchronized views of MPI-IO and POSIX levels. Users can zoom into time intervals or subsets of ranks and inspect detailed attributes of requests, enabling deeper insights into transformations and optimizations applied by

the I/O stack. For large-scale runs with millions of requests, *Drishti* supports time sliced plots to maintain clarity.

1.2.1.3 Automatic Detection of I/O Bottlenecks. *Drishti* automatically identifies common I/O issues—including stragglers, small or redundant requests, workload imbalance, and metadata overhead—and classifies them into four categories based on severity and confidence. It evaluates more than 30 triggers across STDIO, POSIX, and MPI-IO, and highlights detected bottlenecks directly on interactive, multi-layered plots while also generating structured reports with recommendations. This integration transforms raw logs into actionable insights, enabling users to both locate and understand I/O bottlenecks in context.

Co-author Acknowledgement Chapter III contains published material with co-authorship. Specific contributions are detailed in the beginning of the chapter, but co-authors include Dr. Jean Luca Bez, Dr. Suren Byna, and Dr. Boyana Norris.

1.2.2 Cross-Layer I/O Profiling and Source Code Analysis for Bottleneck Diagnosis. While visualization and analysis provides critical insights into how applications interact with the I/O stack, it alone cannot always fully explain the causes of observed bottlenecks. To further answer **RQ 1**, particularly **RQ 1.2**, the Chapter IV of this dissertation extends *Drishti* by combining metrics from multiple sources—such as Darshan, DXT, and Recorder—into a cross-layer view and further drilling down to the application's source code.

1.2.2.1 Cross-Layer I/O Profiling. *Drishti* demonstrates how metrics and traces collected from different tools can be combined into a unified cross-layer perspective. This approach provides visibility into how I/O operations propagate through multiple levels of the stack, making it possible to identify complex performance issues such as stragglers, blocking accesses, or file system inefficiencies.

By integrating diverse data sources into a cohesive view, *Drishti* enables more precise diagnosis of bottlenecks than single-layer analyses.

1.2.2.2 Source Code Analysis. *Drishti* also introduces methods for connecting these cross-layer observations back to the application's source code. Through backtrace capture and mapping to function names and line numbers, *Drishti* allows developers to directly associate low-level I/O inefficiencies with specific code regions. This capability is crucial for turning performance analysis into actionable insights, enabling targeted optimizations at the source level rather than relying solely on system-level tuning.

Co-author Acknowledgement Chapter IV contains published material with co-authorship. Specific contributions are detailed in the beginning of the chapter, but co-authors include Dr. Jean Luca Bez, Yankun Xia, and Dr. Suren Byna.

1.2.3 Runtime I/O Optimization Through a Lightweight End-to-End Workflow. While *Drishti* effectively addresses one of the major limitations in the state of the art—the disconnect between I/O tracing and optimization—the second limitation remains unresolved: the high tuning overhead associated with I/O optimization. To address this challenge and **RQ2** in particular, this dissertation introduces a fully runtime, end-to-end workflow, realized in an implementation named *SmartIO*, that integrates three key components: (1) prediction of future I/O calls through detection of recurring patterns using context-free grammars (CFG), (2) extraction of I/O access patterns and insights from the predicted calls, and (3) optimization via a rules-based mapping engine constructed through literature review and empirical evaluation. This design enables dynamic optimization of critical HDF5, ROMIO, and Lustre parallel file system (PFS) parameters at runtime, eliminating the need for prior profiling, model training, or exhaustive parameter searches.

1.2.3.1 Building the End-to-End Workflow.

HPC applications typically execute periodic I/O phases (e.g., checkpointing or data loading) that account for most of the I/O time [132]. *SmartIO* leverages this by learning I/O behavior during an initial phase, then applying optimizations in subsequent, recurring phases. When a new phase starts, the system determines whether it matches a prior pattern—if so, optimizations are applied; if not, prediction and extraction are re-invoked. *SmartIO* detects the beginning of an I/O phase by monitoring runtime function calls. After identifying a phase, *SmartIO* determines whether it is new or recurring. New phases invoke prediction and extraction, while recurring ones reuse insights from earlier runs. Once a phase is recognized as recurring, the optimization component applies rules-based tuning for HDF5, ROMIO, and Lustre.

1.2.3.2 Major Components.

The following sections provide a brief overview of the three major components of *SmartIO*, with a detailed description presented in Chapter V.

Prediction: The prediction component leverages recurring I/O patterns in HPC applications to anticipate future I/O calls. Building on Recorder's context-free grammar (CFG) mechanism, it detects repetitive call sequences and uses them to forecast upcoming operations in real time.

Extraction: Once future I/O calls are predicted, the extraction component derives key insights such as access modes, transfer sizes, and collective vs. independent I/O behavior. It analyzes call signatures and, where necessary, coordinates across processes to capture a unified view of application behavior.

Optimization: The optimization component translates extracted insights into runtime tuning of HDF5, ROMIO, and Lustre parameters. Guided by rules distilled from best practices and empirical studies, it dynamically selects parameters such as alignment, transfer mode, and striping to improve performance without requiring prior training or manual intervention.

Co-author Acknowledgement Chapter V contains published material with co-authorship. Specific contributions are detailed in the beginning of the chapter, but co-authors include Dr. Chen Wang, Dr. Hariharan Devarajan, Dr. Hank Childs, and Dr. Kathryn Mohror.

In summary, this dissertation is motivated by two persistent limitations in parallel I/O research and practice: (1) the disconnect between profiling/tracing and actionable optimization, and (2) the significant overhead of tuning parallel I/O at scale. To address these challenges, the subsequent chapters are organized around two complementary thrusts. Chapter II surveys the state of the art to situate these challenges in the broader research landscape. Chapters III and IV present solutions that bridge the gap between I/O characterization and optimization through interactive visualization, cross-layer I/O analysis, and source-level analysis. Chapter V then introduces the runtime workflow, discussed previously, to minimize tuning overhead. Together, these contributions form an end-to-end approach for understanding and optimizing I/O in HPC systems.

CHAPTER II

STATE-OF-THE-ART IN PARALLEL I/O CHARACTERIZATION AND EVALUATION

This chapter draws on both unpublished and published material. §2.1, §2.2, §2.3, and §2.4 incorporate material from my Area Exam, which surveyed the state of the art in this research field. For the Area Exam, I was the sole author of the text and completed all writing, while receiving guidance, feedback, and suggestions from my dissertation committee members—Dr. Hank Childs, Dr. Allen D. Malony, and Dr. Jee Choi. A revised version of the Area Exam is also available as a preprint on arXiv, co-authored with Dr. Jean Luca Bez, Dr. Chen Wang, Dr. Hank Childs, Dr. Allen D. Malony, and Dr. Suren Byna.

2.1 Introduction

In recent years, there have been significant advances in parallel processing hardware, which has propelled HPC systems to perform computations at a much more massive scale [10]. These computations also bring a vast amount of data, which, if not appropriately managed, can add significant performance overhead and impact the application's scalability. Many times, the overhead comes from unoptimized parallel I/O. Therefore, ensuring fast and efficient parallel I/O in large-scale HPC systems is critical.

The parallel I/O stack deployed on large-scale computing systems is complex and difficult to tune. Usually, there is an interplay of multiple factors that impact the performance of data movement between storage and computing systems. Also, each stack layer has many tuning parameters and optimization techniques that can improve application performance. I/O performance optimization is a complex problem due

to the multiple factors that can affect performance, such as the interdependencies among the stack layers.

When applications suffer slowdowns, pinpointing the root causes of inefficiencies requires detailed metrics and an understanding of the stack and the different I/O access patterns. These access patterns, which have been identified through numerous research studies, can have a direct impact on the overall performance of the application, therefore it is crucial to optimize these accesses. Optimization techniques such as collective I/O and data sieving can optimize these access patterns and lead to better performance.

To understand the I/O performance, knowing different workload generation techniques, such as benchmarks and simulations, is essential. Benchmarks play an important role in understanding the I/O performance of large-scale applications without actually incurring the overhead of running these large-scale applications. Understanding how to collect performance measurements using profiling and trace analysis tools of different I/O workloads is crucial in understanding the I/O performance. There are also many statistical, predictive, and replay-based analysis techniques that further analyze and evaluate the data collected by these tools and help in further understanding the I/O performance of the application.

This chapter presents a comprehensive survey of the state of the art in large-scale parallel I/O characterization and evaluation. It begins with an overview of the different layers of the HPC I/O stack and their interactions, followed by a taxonomy of performance evaluation strategies. The chapter then examines the major stages of parallel I/O evaluation, including workload generation, data monitoring and collection, and the analysis techniques and tools used to interpret results. By synthesizing insights from a wide range of research studies, this chapter provides

a structured understanding of existing approaches and addresses the following key questions:

- What are the different layers of the HPC I/O stack?
- What are the different profiling and tracing tools available to characterize the I/O behavior of the application?
- What analysis techniques are HPC researchers applying for characterizing and analyzing the I/O behavior?

The remainder of this chapter is organized as follows. §2.2 reviews the different layers of the HPC I/O stack. §2.3, which forms the core of this chapter, surveys state-of-the-art tools and techniques for characterizing and evaluating parallel I/O. Finally, §2.4 summarizes the key takeaways and concludes the chapter.

2.2 HPC I/O Stack

The complex I/O workloads of serial and parallel applications running on large-scale HPC systems are supported by the HPC I/O stack [15]. The I/O stack is complex and has multiple layers, each with many tuning parameters that can be used to improve the application performance [10, 13]. Fig. 1 shows the multi-layered I/O stack deployed on HPC systems. As can be seen from the figure, the I/O stack has multiple layers, such as the high-level I/O libraries, parallel I/O middleware, low-level I/O libraries, I/O forwarding layer, and the PFS between the applications and storage hardware.

2.2.1 High-level I/O libraries. High-level I/O libraries provide abstractions for data modeling and management, allowing portability and high application performance. Some commonly used high-level I/O libraries include HDF5 [40], ADIOS [68], NetCDF [65], and PnetCDF [66]. The HDF5 technology suite

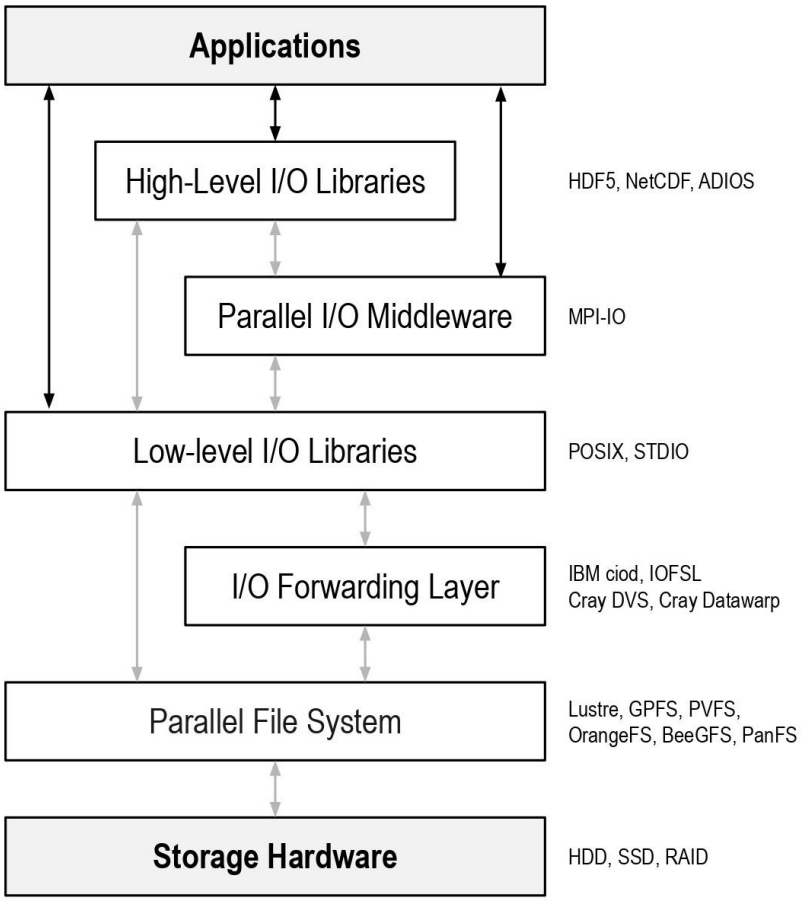


Figure 1. Traditional parallel I/O stack deployed in production scale supercomputers [15]

comprises a data model, a library, and a file format to store and manage data [40]. HDF5 is portable and easily extensible, allowing it to support a variety of data types and store and manage complex I/O data. NetCDF (Network Common Data Form) supports creating, accessing, and sharing array-oriented scientific data using software libraries and machine-independent data formats [100]. The NetCDF-4 package can use HDF5 for storing the data while using the original NetCDF library. PnetCDF uses parallel I/O techniques to provide a high-performance and efficient interface to access NetCDF files. The Adaptable IO System (ADIOS) provides an efficient and flexible solution for researchers and scientists to describe the data in their code as an

external to-the-code XML file for processing outside the running simulation. All of these libraries mentioned above map the abstractions of applications' data and encode the applications' data into portable file formats. They also allow the user to describe the data and the different structures in the file in the form of metadata. Apart from these high-level I/O libraries, there are also some domain-specific I/O libraries as well such as FITS [47, 137] and ROOT [5], which are used for High-Energy Physics (HEP) and astronomy.

2.2.2 Parallel and low-level I/O libraries. The parallel I/O middleware and low-level I/O libraries include MPI-IO (Message Passing Interface I/O) [30], POSIX I/O (Portable Operating System Interface I/O) [1], and STDIO (Standard Input and Output) interfaces. Users can also use these interfaces directly to perform I/O to the file systems. MPI-IO provides a low-level interface for performing parallel I/O. In MPI-IO, a file is defined as an ordered collection of typed data items. Using these typed data items, MPI-IO allows the user to define data models for their applications. It also provides two types of I/O calls which are independent and collective I/O. Independent MPI I/O calls are defined as those types of calls made by any subset of the processes participating in I/O, with each process handling its I/O independently. Some of the basic independent MPI I/O calls are `MPI_File_read()` and `MPI_File_write()`. Collective MPI I/O calls are those I/O calls that are made by all processes participating in a particular I/O sequence. The basic collective calls are `MPI_File_write_all()` and `MPI_File_read_all()`. POSIX I/O, on the other hand, views a file as a sequence of bytes. Through this interface, contiguous regions of bytes can be transferred between the file and memory, and non-contiguous regions of bytes can be transferred from memory to a file by giving complete low-level control of the I/O operations. The major drawback of POSIX I/O in the context of HPC

is that it supports very little parallel I/O. For example, ensuring collective access to files is not easy in POSIX and the user has to manually coordinate access and ensure consistency. It is also not flexible in terms of file metadata as it prescribes a specific set of metadata that a file must possess [96]. STDIO in contrast, provides abstractions to deal with the stream of input and output bytes. It is comprised of the C stdio.h family of functions, such as `fopen()`, `fprintf()`, and `fscanf()`. These I/O functions are commonly used in many parallel I/O applications; however, STDIO functions do not directly support random access to data files. It relies on the user to create an input/output stream, seek the position in the file from where to read or write, and then read/write bytes in sequence from/to the stream.

2.2.3 I/O forwarding layer. The I/O forwarding layer aims to reduce the number of clients concurrently accessing the PFS by providing an additional transparent layer between the compute nodes and the data servers in the form of I/O nodes. Instead of the applications directly accessing the PFS, requests are first received by these I/O nodes which forwards these requests to the PFS in a manageable way. This technique allows a smooth flow of I/O requests, providing a layer between the application and file system, and makes possible certain optimization techniques such as aggregation, request scheduling, and compression. The I/O forwarding layer technique was initially introduced for the Blue Gene/L system [3], but now has been extended to many of the top 500 supercomputers. One such example is the Tianhe_2 supercomputer which has 256 intermediate I/O nodes. These nodes, powered with high-speed SSDs, have configurations for each file which determines when data is transferred from the I/O nodes to the PFS.

2.2.4 Parallel File System. Large-scale HPC systems rely on PFS to provide an infrastructure for persistent shared storage and a global namespace across

distributed storage servers to read and write to files. A PFS mainly has two types of servers: the data server and the metadata server. Metadata servers are responsible for handling the file metadata, which is information related to the size, permissions, and location of the file on the data servers. Before any client can access the data in a file, they must obtain the layout information of the file from the metadata. Accessing the metadata can also add some overhead, so some systems cache the metadata on the client side for faster access. However, that can lead to issues such as cache coherence, especially when many clients are concurrently accessing the PFS. On the data server, the files are distributed using data striping [113]. In the technique, the file is divided into fixed-size chunks called stripes, and the stripes are distributed to the servers in a round-robin approach. A PFS can retrieve stripes from different servers in parallel, hence increasing throughput. Different machines have different stripe sizes for example the Google File System has a stripe size of 64MB. PVFS [23], Lustre [150, 43], and GPFS [106] have default stripe sizes between 64KB and 1MB. To keep consistency when having concurrent accesses, some PFS systems use locking on the server. Lustre is one such system that uses stripe granularity, i.e., multiple clients can not access the same stripe concurrently. However, other systems, like PVFS, allow the users to deal with consistency for simplicity and better performance. PFS systems also provide mechanisms for fault tolerance which include measures such as replication of data and metadata. This is achieved by keeping mirrored servers. There can be a performance overhead for copies of the same data updated across the mirrored servers however having the same data on more than one server can improve performance by allowing parallel access. Some popular file systems include Lustre, PVFS, IMB GPFS, and the Panasas file system [136].

2.2.5 Storage Hardware.

There are a variety of storage hardware used in a supercomputer. HDDs [84] are a popular storage medium that has been in use for many years now. They are made up of magnetic-surfaced rotating platters. To access any data in an HDD, a seek operation has to be performed in which the head has to be moved to the proper location. HDDs give the best performance when the underlying data needs to be accessed sequentially instead of randomly, as that reduces the overhead of the seek operation [91]. A recent flash-based alternative to hard disks is SSDs. SSDs have higher bandwidth and less overhead for sequential access than HDDs.

2.3 Parallel I/O Characterization and Evaluation

In this chapter, we first provide a taxonomy, as depicted in Fig. 2, for the iterative process of evaluating and optimizing parallel I/O. The I/O evaluation and optimization mainly consists of three major phases: (1) Measurements and Statistics Collection, (2) Modeling and Prediction, and (3) Simulation [86]. For the scope of this chapter, we will only be looking at (1) Measurements and Statistics Collection and (2) Modeling and Prediction. We survey different research works and tools for each phase and present our results in the sections below.

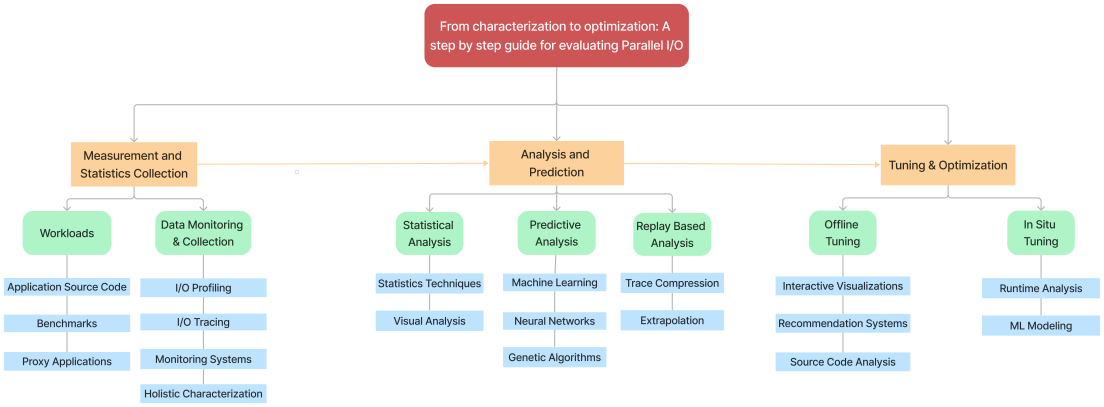


Figure 2. Taxonomy of different phases in parallel I/O evaluation [86]

2.3.1 Measurement and Statistics Collection.

In this phase, we look at different methods for generating I/O workloads through real-world data or benchmarks. Then, we look at some tools that can be used to collect statistics and information about the I/O behavior of these workloads.

2.3.1.1 I/O Workloads.

The first step in parallel I/O evaluation is generating and characterizing I/O workloads. Different workload generation techniques are available, some discussed in the sections below.

Application Code The application code is the most accurate workload that can be used for characterization purposes. However, many times, the application code is not accessible or is too huge or long to be profiled for its I/O behavior. If the application code is unavailable, benchmarks can be used to generate synthetic workloads for characterization.

Benchmarks Benchmarks simulate the workflow and access patterns of the original application, allowing performance evaluation and tuning of the original application without actually incurring the overhead of running it. In this section, we study different system and application benchmarks. System benchmarks use different interfaces and access patterns to benchmark parallel file and storage system performance, whereas application benchmarks simulate different access patterns to benchmark application performance.

We study different benchmarks available out there in this section and categorize them based on whether they are parallel file and storage system benchmarks or application benchmarks. For parallel file and storage system benchmarks, we study different characteristics of the benchmarks such as data, metadata, GPU support, etc. For application benchmarks, we study different characteristics such as the layers

of the I/O stack these benchmarks cover and the I/O access patterns they generate. These findings are summarized in Table 1 and 2.

Table 1. System-Level I/O Benchmark Comparison

Benchmark	Parallel	Metadata	R/W	GPU	UI	Interface	I/O Focus
IOR [108]	✓	✗	✓	✗	✗	POSIX, MPI, HDF5	Seq I/O, scalable
MDTest	✓	✓	✗	✗	✗	—	Metadata ops
MD-Workbench	✓	✓	✗	✗	✗	—	Metadata stress
fio	✗	✗	✓	✗	✗	POSIX, Block	Rand/seq I/O test
Elbencho	✓	✓	✓	✓	✓	POSIX, MPI, GPU	Mixed, GPU-aware
IOzone	✗	✗	✓	✗	✗	POSIX	Pattern diversity
Bonnie++	✗	✓	✓	✗	✗	POSIX	Local FS eval
OBDFilter-Survey	✗	✗	✓	✗	✗	Lustre	OST benchmarking
IOBench [139]	✗	✗	✓	✗	✗	POSIX, AIO	Device/protocol

Table 2. Application-Level I/O Benchmark Comparison

Benchmark	Parallel	Format	I/O Mode	Configurable	MPI I/O	Collective	Primary Focus
h5bench [16]	✓	HDF5	R/W	✓	✓	✓	Extensible kernels, scientific HDF5
DLIO [33]	✓	POSIX	R/W	✓	✓	✗	AI/DL I/O pipelines
HACC-IO [6]	✓	POSIX	Write	✓	✓	✓	N-body checkpointing
FLASH I/O [64]	✓	HDF5	Write	✗	✓	✓	Structured HDF5 writes
b_eff_io [97]	✓	MPI-IO	R/W	✓	✓	✓	Bandwidth stress test
MPI Tile I/O [102]	✓	MPI-IO	R/W	✓	✓	✓	Noncontiguous tiled patterns
NetCDF-Bench [6]	✓	NetCDF	R/W	✗	✓	✓	Climate model pattern emulation
Parabench [82]	✓	POSIX	R/W	✓	✓	✓	Programmable synthetic patterns

Workload Replication and Simulation Frameworks Proxy applications are small and simplified codes of large and complex production applications that encapsulate the important features of these applications without forcing the user to assimilate large and complex code bases. Messer et al. [80] presents proxy applications called MiniApps, which are based on large-scale application code at the Oak Ridge National Lab (ONRL). Encapsulating all the important features and details of these large-scale applications, these MiniApps run on production systems at ORNL. Dickson et al. [35] is another work that replicates five different I/O workloads using MACSio [45] proxy application and Darshan [21].

Durango [24] generates scalable workloads from real applications using the performance modeling language Aspen and the HPC CODES framework. Snyder et al. [111] presents IOWA, a novel I/O workload abstraction for generating diverse I/O workloads based on the inputs sources. Dickson et al. [34] extracts the I/O pattern of the application and generates a suitable proxy application.

2.3.1.2 I/O Monitoring and Collection. Once we have the I/O workload available, the next step in I/O performance evaluation is I/O data monitoring and collection. This includes collecting I/O performance data of the application, such as time spent in I/O operations, read and write throughput, and total execution time. This data is then used to characterize the I/O performance of the application and further use it for analysis, modeling, and prediction.

There are a variety of I/O monitoring and collection tools available out there that help understand the I/O behavior of the application by looking at the interaction between the application workload and the underlying I/O stack. These tools can be divided into two categories based on how they collect the performance data. This performance data can be of two types: `traces` and `profiles`. `Profiles` store critical I/O information and statistics, such as file access patterns, number of floating point operations performed, number of function invocations, average execution time of a function, etc. `Traces`, on the other hand, provide a more fine-grained view of the I/O performance of the application, reporting data like timestamps of function calls along with their chronology. Because of their detailed metric collection, `traces` increase the overhead for I/O monitoring, degrading overall system performance.

This section will discuss the HPC I/O profiling and tracing tools available to the HPC community. More specifically, we will examine how these tools work and which layer of the HPC I/O stack they characterize. We will also look at some

storage-system-level monitoring tools available to characterize the I/O behavior of the storage system. Lastly, we will also look at some recent monitoring systems that provide an all-encompassing and cohesive view of the application's I/O behavior.

I/O Profiling and Tracing Tools *Darshan* [21, 112] is a commonly used application I/O characterization tool deployed on many large-scale HPC systems. It collects fine-grain trace data for POSIX and MPI-IO layers (parallel and low-level I/O layer) and basic data for high-level I/O libraries such as HDF5 and PNetCDF. There have been recent advances in *Darshan* [112] to add support for tracing PFSs as well, such as Lustre.

Implemented as a set of user-space libraries, *Darshan* does not require any source code modification to instrument the application. Instead, it can be added to the application either statically or dynamically. Dynamic executables use the `LD_PRELOAD` environment variable to inject *Darshan* instrumentation in the application, whereas static executables link *Darshan* to the application during the linking phase. Instead of working as a typical tracing tool that logs every I/O operation, a bounded amount of data for each file is collected by *Darshan*, allowing compact storage of data. This data contains metrics such as I/O operations count, I/O access sizes, timers, and other statistical data relevant to the I/O performance of the application. Here are some of the metrics collected for each layer [21]:

- POSIX: read, write, open, seek, stat, mmap, fopen
- MPI-IO: collective, independent, split, or nonblocking read and write
- MPI-IO datatypes and hints
- Access type: unaligned, sequential, consecutive, and strided access

– Timestamps for open, close, first and last I/O

– Total bytes read and written

– Total time spent within POSIX and MPI-IO I/O operations

– Histograms of access, stride, datatype, and extent sizes

Fig. 3 shows how *Darshan* instruments an application. During the application execution, *Darshan* instruments different layers of the I/O stack for each process, generating data records characterizing the application's I/O workload and writing that data to the process's file while deferring all communication and I/O operations until the application terminates. When the application terminates, as part of the `darshan_shutdown` routine, it performs data aggregation and compression, which comprises aggregating and compressing the data of all the file-per-process into a single file. *Darshan* does this process in parallel and uses MPI-IO collective writes to make this step faster. This ability to defer communication and I/O until application shutdown, coupled with the bounded data collection, makes *Darshan* lightweight and portable, allowing seamless deployed on large-scale HPC systems such as the Argonne Leadership Computing Facility (ALCF), the National Energy Research Scientific Computing Center (NERSC), and the National Center for Supercomputing Applications (NCSA).

The overhead introduced by *Darshan* [21] is negligible. Fig. 4 shows the overhead of different versions of *Darshan* on the IOR [108] benchmark. The box plots indicate the distribution of I/O times in each case (with or without *Darshan*). It can be seen from the figure that the overhead when *Darshan* is enabled is very minimal and has very little impact on application performance.

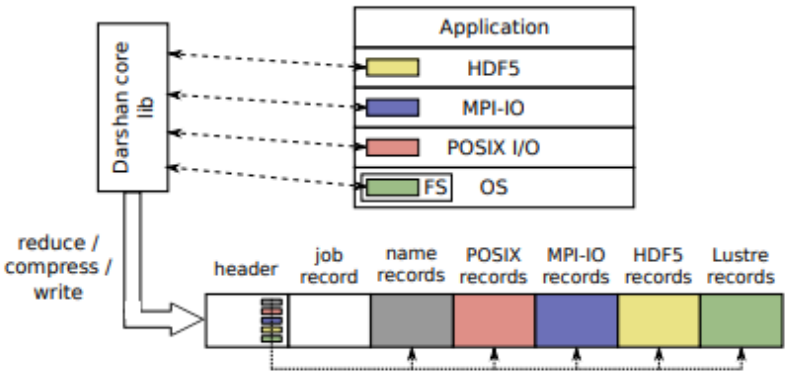


Figure 3. High level overview of *Darshan's* architecture [112]

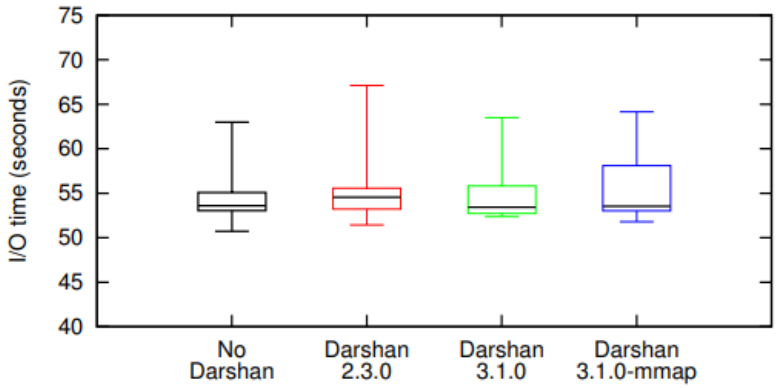


Figure 4. Impact on application performance with Darshan enabled [112]

Darshan eXtended Tracing (DXT) [143] extends *Darshan* to provide fine-grain traces of the application I/O to study behaviors of a wide range of workloads in greater depth. The DXT module is disabled by default and can be switched on using an environment variable. The overhead introduced by this module is also minimal (around 1%). DXT logs can be parsed, analyzed, and visualized offline, allowing this analysis on any system. This analysis can detect issues like workload imbalance, lock contention, and stripe misalignment.

Recorder [131] is a multi-level I/O tracing framework that captures I/O information from multiple levels of the I/O stack, including HDF5, MPI-IO, and POSIX. It requires no source code changes, and the user controls which stack layers can be traced. It uses function interpositioning to prioritize itself over the standard functions using library preloading in Linux. Once *Recorder* is specified as a preloaded library, it intercepts HDF5, POSIX, and MPI-IO calls and reroutes these requests to the tracing routine. The tracing routine saves the function name, file name, entry timestamp, and the function parameters and then calls the original function and records its exit timestamp. All of this information is converted into a trace record and then compressed. Fig. 5 shows some of the features of the *Recorder* framework. Wang et al. [131] is a newer version of *Recorder* and adds new features such as trace format optimization, visualizations, and improved tracing for POSIX.

IOPin [56] is a dynamic instrumentation tool developed to analyze the performance of parallel I/O. It uses Pin [26], which is a lightweight binary instrumentation tool to intercept the MPIIO and PVFS layers. Because of this, *IOPin* does not require recompilation of the source code or the I/O software stack. Two Pin profiling processes on the client and server sides generate trace information at the border of the MPI and PVFS layers. The client-side Pin process contains

Feature	Recorder
Parallel file system compatibility	Yes
Ease of installation and use	1 (V. Easy)
Anonymization	3 (Medium)
Event Types	System calls and Library calls
Control of trace granularity	Moderate
Replayable trace generation	Planned
Trace replay fidelity	N/A
Reveals dependencies	No
Intrusive vs. Passive	1 (V. Passive)
Analysis tools	Planned
Trace data format	Binary and Human Readable
Accounts for time skew and drift	No
Elapsed time overhead	Under investigation

Figure 5. Features of the *Recorder* framework [131]

information such as the MPI call ID, rank, PVFS call ID, I/O type (read/write), and latency. The server-side Pin process stores information such as PVFS server ID, latency in the server, bytes to be read/written, the number of disk accesses, and disk throughput for the MPI I/O call at runtime

TAU [109, 110] is a portable and flexible integrated toolkit of performance instrumentation and analysis of HPC applications. Although *TAU* is not I/O specific profiling tool, it does provide different instrumentation techniques to analyze the I/O performance of the application. One of these approaches is pre-processor-based instrumentation, in which the header files are redefined to instrument POSIX I/O calls. It also performs MPI instrumentation using the PMPI interface

Score-P [59] is a performance measurement infrastructure for profiling, event trace recording, and online analysis of High-Performance Computing (HPC) applications. *Score-P* provides different ways to instrument the application code, either through automatic compiler instrumentation or manual instrumentation. In parallel I/O, *Score-P* can instrument POSIX I/O and MPI I/O routines.

Mistral [81] is a commercially available HPC I/O monitoring tool from Altair. *Mistral*, due to its lightweight, can be easily deployed on production systems and is flexible enough to monitor I/O patterns on large-scale HPC and cloud systems. It uses files called contracts to store the rules for monitoring and throttling I/O. These contracts can be modified at runtime by privileged users. With its system and storage agnostic features, *Mistral* can monitor each job and file system's read, write, metadata use, and storage performance. Below is a list of data that Mistral collects.

- Collect I/O data per job, user group, mount point, and host
- Track read(), write(), and metadata operations
- Track time spent in I/O for each application to get file system performance
- Track statistics like the total, mean, and max time spent doing I/O per job
- Tracks CPU and memory utilization

Ellexus Breeze is another flexible and user-friendly offline analysis tool to help optimize and tune complex Linux applications. It can also capture MPIIO and POSIX calls and store the information related to them in binary trace files. *Ellexus Breeze* classifies I/O calls into three different categories:

- Bad I/O: Small read and write, Zero-byte I/O, Backward seek etc.
- Medium I/O: Large read and write, Forward seek, Delete operations, etc.
- Good I/O: Medium-sized read and write, etc.

IOPro [57] is a performance analysis and visualization framework for parallel I/O HPC applications. *IOPro* is different from the tools mentioned above in this way that

Tool	Profiling	Tracing	Monitoring	High Level	Parallel and Low Level	PFS	Storage
Darshan [21]	✓	✗	✗	✓	✓	✗	✗
IOPin [56]	✓	✗	✗	✗	✓	✓	✗
IOPro [57]	✓	✗	✗	✓	✓	✓	✗
IPM [140]	✓	✗	✗	✗	✓	✗	✗
TAU [109]	✓	✓	✗	✓	✓	✗	✗
Score-P [59]	✓	✓	✗	✗	✓	✗	✗
Darshan DXT [143]	✗	✓	✗	✗	✓	✗	✗
Recorder [131]	✗	✓	✗	✓	✓	✗	✗
RIOT I/O [141]	✗	✓	✗	✗	✓	✗	✗
IOscope [101]	✗	✓	✗	✗	✗	✗	✓
ScalaIOTrace [128]	✗	✓	✗	✓	✗	✗	✗
Mistral/Breeze [81]	✗	✗	✓	✗	✓	✗	✗
DKRZ Monitoring [62]	✗	✗	✓	✗	✗	✓	✓
LLview [41]	✗	✗	✓	✗	✗	✓	✓
GUIDE [127]	✗	✗	✓	✗	✗	✓	✓
LMT [150]	✗	✗	✓	✗	✗	✓	✓

Table 3. I/O profiling, tracing, and monitoring tools and their characteristics

it can instrument the complete parallel I/O stack. Instead of manually instrumenting each layer of the stack, *IOPro* takes the I/O software stack as input and generates an instrumented I/O stack to trace specific I/O functions and individual components across each layer. A high-level framework of *IOPro* is shown in Fig. 6

As can be seen from the framework in Fig. 6, *IOPro* has three main components: instrumentation engine, execution engine, and data processing engine. The instrumentation engine consists of a probe selector and a probe inserter. It inserts these probes into the I/O software stack automatically and generates an instrumented version of the software stack. The execution engine is responsible for building and compiling the instrumented software stack. The data process engine collects all trace log files from each layer of the instrumented I/O stack.

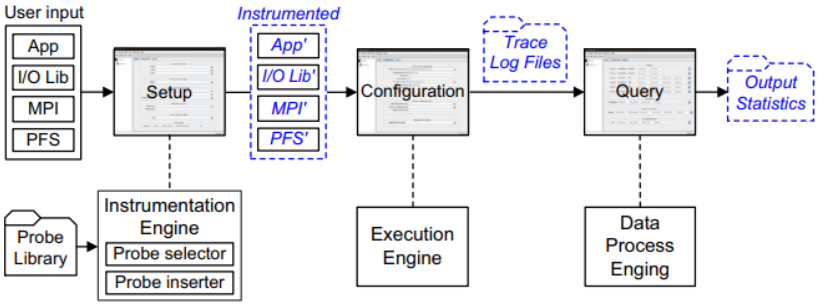


Figure 6. Overview of the *IOPro* framework [57]

IOscope [101] is a flexible I/O tracer for characterizing the I/O patterns of the storage systems' workloads. It relies on the *extended Berkeley Packet Filter (eBPF)* technology to instrument the kernel and communicate the target kernel data towards a userspace process. The I/O pattern which the tool detects is the order in which the I/O requests access the file offsets during accessing on-disk data. *IOscope* consists of two tools: *IOscope classic* and *IOscope mmap*. The main idea is to trace and filter the workloads' I/O requests to construct the workloads' I/O patterns.

RIOT I/O [141] is another I/O tracer that uses function interpositioning to intercept the POSIX and MPI-IO libraries storing timing and other information about each function call. Similarly, *ScalaIOTrace* [128] is another tracing and analysis tool that collects trace data across multiple layers in the I/O stack. It also provides novel capabilities to analyze and collect statistical information from the collected traces automatically. *IPM* [140], developed at Lawrence Berkeley National Lab, is a portable profiling tool that profiles parallel programs' performance aspects and resource utilization. It uses an interposition layer to catch all the calls between the application and the file system layer.

The German Climate Computation Centre (DKRZ) developed a monitoring system [62] to develop the workload of the Mistral Supercomputer. This flexible

and extensible monitoring system gathers statistics from 3340 client nodes, 24 login nodes, and 2 Lustre file systems. This system comprises open-source components such as Grafana and a self-developed data collector. It provides statistics about the login nodes, user jobs, and the workload manager queue. Initially, the monitoring system gives the users an overview of the system's current state, providing information such as the current load of login nodes and the number of used nodes on Slurm partitions. For each node in the system, the monitoring system collects metrics such as CPU usage, memory usage, luster file system usage, and energy consumption.

The Julich Supercomputing Centre (JSC) developed *LLview* [41] in 2004 to provide an interactive graphical tool to monitor jobs running on different workload managers such as SLURM. It has a job reporting module, which provides detailed information about all the individual jobs running on the system. The way *LLview* works is that it interacts with the different sources in the system to collect performance data and aggregate that data to present to the user in a web portal. Apart from providing a live view of the system, this web portal provides a list of the jobs running on the system in the form of a table, with each row showing the aggregated information for that job, such as the owner, project, start time, end time, etc. Upon clicking on a job, the user can see timeline graphs for the key performance metrics. All metrics in job reporting are gathered minutely, and the detailed reports can be made available offline as well in the form of a PDF.

GUIDE (Grand Unified Information Directory Environment) [127] is another framework that has been developed to collect, federate, and analyze log data from the Oak Ridge Leadership Computing Facility (OLCF). First, the data is extracted, and logs are collected. The framework collects data at each level of OLCF subsystems. For example, at the storage subsystem level, data is collected for the disk layer (the

2,016 OSTs, encompassing the 20,160 disks), the redundant RAID, controllers (72), the OSSes (288), and the Lustre PFS level. In the next step, the data is preprocessed using data cleansing techniques. In the federation step, the data is federated in a scalable repository to make post-processing and visualization of the data easier. Lastly, in the post-processing step, a suite of analytics and post-processing techniques are applied to get insights from the data.

Lustre Monitoring Tool (LMT) [150] is an open-source tool that monitors Lustre activity on HPC systems and presents a MySQL database that contains aggregated Lustre-specific counters on each object storage server (OSS) and metadata server (MDS). LMT provides information such as bytes read/written, CPU load averages, and metadata operation rates. Similarly, *ggiostat* is another tool to collect similar data from IBM Spectrum Scale file systems. Developed at the Argonne Leadership Computing Facility (ALCF), it provides information such as bytes read/written and reads, writes, and metadata operations count retrieved from the server and client clusters.

Holistic Characterization Tools Until now, we have looked at different I/O characterization tools and storage-server-level monitoring tools that instrument and monitor different layers in the I/O stack. These tools collect performance data for individual components in the I/O stack, but looking at this individually does not convey the whole picture of the application's I/O performance. In many cases, the data from different components and layers of the stack needs to be combined to get a more holistic view of the application's I/O behavior. This often requires involving an I/O expert in the loop to translate this disparate data, but not only is this practice labor-intensive, but it is also not sustainable as the system grows and scales. To tackle this issue, frameworks and monitoring systems have been developed that combine the

insights from the performance data collected by various component-level monitoring tools and provide a holistic view of performance across the entire I/O stack. We will look at some of these frameworks in this section.

The Total Knowledge of I/O (TOKIO) [74] is a framework that connects different component-level monitoring and characterization tools, combines their insights, and presents a single, holistic view of the I/O performance across the entire I/O stack, which static analysis tools and user interfaces can further use. The distinguishing factor in the TOKIO framework is that it provides a modular implementation that connects to whatever monitoring and profiling tools are available on the HPC system. It mainly collects data from the tools that profile application behavior such as Darshan and tools that look at storage system and network traffic and health.

Connectors are the foundational layer of TOKIO. These modular and independent components provide an interface to connect with individual component-level tools, providing data from these tools. On top of connectors are *TOKIO tools*, which make this data collected by the connectors semantically closer to how analysis applications would like that data to be. These tools provide abstractions to hide the complexities of the underlying data source. High-level applications like command line tools, statistical and data analysis tools can easily connect with TOKIO interfaces to analyze the holistic data collected.

Similar to TOKIO, the Unified Monitoring and Metrics interface (UMAMI) [72] also provides a normalized, holistic view of the I/O behavior of the application. With the component-level data already being collected by different tools at the application and storage-system levels, this data is analyzed over one month, and the changes in the metrics are shown in UMAMI. Luu et al. [77] presents a multiplatform study in which Darshan logs representing a combined total of six years of I/O behavior of a

million jobs across three leading HPC systems are mined and analyzed. It studies the evolution of the I/O behavior of the application over time, and based on the findings, the study provides techniques to improve the I/O performance of an application.

Beacon [145] is an end-to-end I/O monitoring and diagnosis tool developed for the TaihuLight supercomputer. Beacon monitors I/O for different nodes such as the compute, I/O forwarding, metadata, and storage nodes. At each node, Beacon deploys a daemon to monitor I/O and collect performance data for later aggregation. All this aggregated data is stored in JSON objects, which are then used to build profiling and analysis services such as detecting anomalies, per job I/O performance, etc.

2.3.2 Analysis, Modeling, and Prediction. Once the profile and trace data are collected for the HPC I/O application using the different I/O profiling, tracing, and monitoring tools that we have discussed in the previous section, the next step in parallel I/O evaluation is to analyze this data to identify I/O issues, model I/O performance, and predict future I/O performance of the HPC system. In this section, we look at different tools and research work that has been done to analyze the I/O performance of an HPC system. This work can be divided into three main categories: Statistics and Analysis, Predictive Analysis, and Replay-Based Modeling.

2.3.2.1 Statistics and Analysis. The traditional way of analyzing I/O data is through applying statistics and analysis techniques to extract meaningful patterns from the I/O traces, I/O characterization profiles, and other logs. Different statistical techniques such as Arithmetic Mean, Standard Deviation, Linear Regression, Probabilities, etc. are applied to the data to classify, correlate, and extract meaningful patterns. Applying statistical analysis on HPC workloads and data requires in-depth system knowledge and extensive human effort.

Research/Framework	Analysis Technique	Analysis Goal
Yildiz et al [146]	Evaluates point of contentions in HPC storage system	Identify root causes of I/O interference
Lockwood et al [73]	Correlative and financial market technical analysis on year-long performance data	Identify various I/O performance issues
IOMiner [134]	Centralized storage schema that combines log data of different instrumentation tools	Detect root causes of poor I/O performance such as small I/O, non-collective I/O etc.
Luu et al [77]	Statistical analysis of Darshan logs spanning years	Study the I/O behavior of applications on three large-scale supercomputers
Wan et al [130]	Analyzes Recorder trace data	Study performance and variability of the storage system and provide feedback
pytokio [74]	Analysis routines and connectors to extract insights from monitoring tools	Holistic analysis of the I/O performance of parallel systems
GUIDE [127]	Analyzes log data from the Oak Ridge Leadership Computing Facility (OLCF)	Understand the performance of the storage system

Table 4. Statistical analysis tools/research and a brief description of what kind of analysis they perform

Yildiz et al. [146] presents an extensive experimental study exploring the root causes of I/O interference in HPC systems. It first identifies the points of contention in an HPC storage system and evaluates how the application's access patterns, file system and network configuration, and storage devices affect interference. Based on the experiments on the Grid'5000 testbed and the OrangeFS file system, the study highlights seven root causes of I/O interference. Lockwood et al. [73] investigates various I/O performance issues by studying and analyzing performance data collected over a year at two leadership high-performance computing centers. The study looks at transient and long-term trends in I/O performance variability using different analysis techniques, such as correlative analysis and financial market technical analysis techniques, on time series I/O performance data to identify regions of interest. The insights provided in this paper can help broaden the scope of instrumentation and analysis tools.

IOMiner [134] is a large-scale analytics framework to analyze instrumentation data using a centralized storage schema that combines log data collected using different instrumentation tools and a sweep-line analysis function that identifies root causes of poor I/O performance of an application. The centralized storage schema is designed to take away any format difference that the different log data might have, making it query-friendly and easy to use for the sweep-line analysis function. One of the challenges that *IOMiner* addresses is how to mine useful information and insights from the performance data collected on supercomputers, which can run as many as millions of jobs quickly. To address this challenge, *IOMiner* uses a Python API for the Spark framework called PySpark, which speeds up data analysis on large-scale systems using parallel processing. In the analysis phase, *IOMiner* provides an

analysis function that looks at five contributing factors to the application's poor I/O performance. These factors are:

- Small I/O requests
- Nonconsecutive I/O requests
- Utilization of collective I/O
- Number of OSTs used by each job
- Contention level

Luu et al. [77] presents a comprehensive study of the I/O behavior of applications on three large-scale supercomputers. It analyzes the Darshan logs on the three different supercomputers, spanning years and months. Through in-depth analysis of these logs, the paper looks at different aspects of the I/O performance of the applications on each supercomputer. For example, one of the paper's analyses is a platform-wide analysis in which the performance of I/O workloads on the three supercomputers is studied, and techniques are presented to identify underperforming apps. The analyses show that most of the apps on each supercomputer never exceed the platform peak I/O throughput, and low I/O throughput is the norm. Fig. 7 shows the maximum I/O throughput of each app across all its jobs on the three supercomputers under consideration, with the horizontal line showing the platform peak I/O throughput. The paper also presents other insights, such as discovering that a few jobs and apps mainly dominate each platform's I/O usage.

Wan et al. [130] presents a comprehensive study and analysis of the I/O performance of a production leadership-class storage system by collecting large amounts of trace data using the *Recorder* [131] tracing tool. After collecting the data,

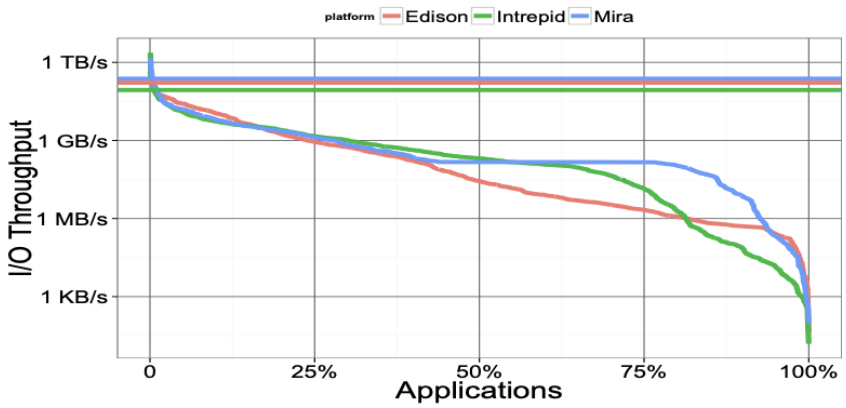


Figure 7. Maximum I/O throughput of each application and the platform peak I/O throughput [77]

the study thoroughly analyzes this trace data to get insights into the performance and variability of the storage system. It provides some feedback on resolving the issues encountered in the analyses. The paper looks at two main scenarios to understand I/O performance do the storage system: 1) *Performance of I/O issued to a single OST*, and 2) *Impact of concurrent access to single OST*. It also looks at other scenarios, such as the effect of system caching on user-perceived performance and the effect of large-scale parallel I/O. Some of the key insights of the paper are:

- Concurrent access to a single OST might not lead to performance degradation for specific write sizes
- To maintain high throughput, system caching is critical
- Interference can be generated along the I/O path because of large scale parallel I/O
- Imbalanced I/O traffic distribution among OSTs can lead to performance degradation

PyTokio [74] is a Python implementation of the TOKIO framework discussed in previous sections. *PyTokio* makes holistic analysis of the I/O performance of parallel systems easier by providing connectors to interface with many commonly used monitoring tools. It also provides analysis routines to extract insights from the data collected from the different monitoring tools. One of the components of *PyTokio* is tokio tools which provides a high-level abstraction for accessing the combined data collected from multiple monitoring tools, allowing different analysis tools to build upon *PyTokio*. GUIDE (Grand Unified Information Directory Environment) [89] is another framework that we discussed earlier, which collects, federates, and analyzes log data from the Oak Ridge Leadership Computing Facility (OLCF). It also provides some analytics and visualizations of the storage system, like looking at the Lustre OST usage over time, I/O block sizes and space efficiency, and I/O request size distribution.

2.3.2.2 Predictive Analysis. Predictive Analysis, as the name suggests, is the kind of analysis that predicts future events based on the current information provided. Such analysis deploys techniques like data mining, predictive modeling, and machine learning on the trace/log data to predict future performance. By building a predictive model using the trace data available, accurate predictions on the future performance of the HPC system can be made. Such an analysis also includes auto-tuning techniques to predict the best parameters for each layer of the HPC I/O stack for optimized performance.

To predict future I/O operations, Omnisc'IO [36] presents a novel approach using a grammar-based model of the I/O behavior of the application. Not only does it predict when future I/O operations will occur, but it also tells where these operations will occur and how data will be accessed. The grammar-based model is built at runtime using an algorithm derived from Sequitur [87]. The way Omnisc'IO operates

Research/Framework	Prediction Technique	Prediction Output
Omnisc'IO [36]	Grammar-based model	Next I/O operation
Sun et al [115]	Random forest approach	Execution Time
Schmidt et al [105]	Artificial Neural Networks	File Access Times
Nemirovsky et al [85]	Six Different Prediction Models[1]	Storage System Performance State
IONET [61]	Deep Neural Networks	Latency of every I/O of a full workload
Isaila et al [53]	Gaussian Process Model	I/O performance sensitivity to application and file characteristics
Behzad et al [8]	Genetic Algorithm	Best combination of parameters for different layers of the I/O stack
Bagbaba et al [7]	Random Forest	Collective I/O performance
Kim et al [58]	Six Different Prediction Models[2]	I/O performance such as read, write throughput

Table 5. Predictive analysis tools along with their prediction technique and output

is that first, it gets the program's call stack, associating each call with an integer called *context symbols*. Then, it builds a grammar-based model from these *context symbols*. Once the grammar is built, it predicts future events by choosing a predictor from the grammar and associating each predictor with a weight. A brief architecture of Omnisc'IO is shown in Fig. 8.

Sun et al. [115] presents a novel framework for modeling and predicting the execution times of MPI programs. The framework captures the syntax tree of the parallel program and automatically instruments it, collecting important features. The instrumentor, developed in clang, inserts detective code around loops, branches, assignments, and MPI communications, generating an instrumented version of the code. Once the features are collected, the goal is to find a correlation between the collected features and the execution time. This multivariate nonlinear regression

[1]Classification and Regression Trees (CART), Naive Bayes (NB), Gradient Boosting (GBT), Support Vector Machines (SVM), Random Forests (RF), and Neural Networks (NN).

[2]Linear, Polynomial, K-nearest neighbors, Gradient boosting random forest (GBDT), Random Forests (RF), Multilayer perceptron (MLP), and Convolutional neural network (CNN).

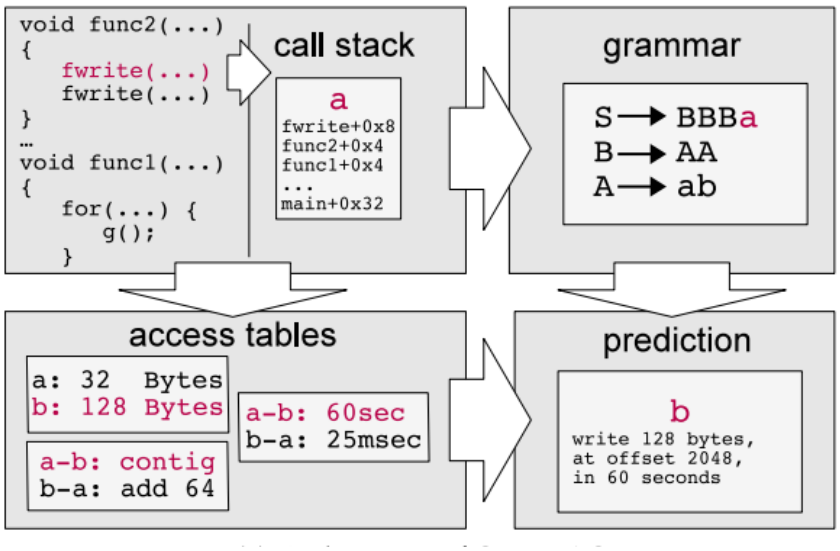


Figure 8. Omnisc'IO Architecture [36]

problem is solved using a random forest approach. Schmid and Kunkel [105] is another research that uses machine learning with artificial neural networks to analyze and predict file access times of a Lustre file system.

Nemirovsky et al. [85] presents a lightweight parallel test harness to collect I/O data on HPC systems and monitor performance states on different storage subsystems. Then, it formulates a machine learning model to predict the transitions between those performance states during runtime. Treating this as a classification problem, this work uses a classifier to predict which I/O state a future I/O operation will likely encounter for an I/O path to an OSS. This work applies six commonly used machine learning classifiers to the dataset: classification and regression trees (CART), naive bayes (NB), gradient boosting (GBT), support vector machines (SVM), random forests (RF), and neural networks (NN). According to the results, SVM provides the best prediction accuracy.

IONET [61] is another ML-based I/O latency predictor that builds models of storage devices. It then uses this model to predict the latency of every I/O of a

full workload running on a target storage cluster without running it on the cluster. It collects traces from various sources and industry partners and then designs an ML model to learn from them. The ML model uses machine learning techniques such as Deep Neural Networks, Random Forest, Logistic Regression, etc., to predict I/O latency from these traces. [53] presents a sensitivity-based modeling framework that predicts the I/O performance of the system and its variability as a function of application and file system characteristics using a Gaussian Process Model.

Behzad et al. [8] optimizes the I/O performance of applications using an autotuning framework. This autotuning framework can efficiently optimize the different layers of the I/O stack, such as HDF5, MPI-IO, and Lustre, without requiring any source code changes. The framework has two main components: *H5evolve* and *H5Tuner*. *H5evolve* uses a genetic algorithm to search the I/O parameter space to find the best parameter combinations and then uses *H5Tuner* to inject the new I/O parameters in the application with minimal user involvement. [7] presents another machine learning-supported I/O autotuning framework to tune I/O parameters for different layers of the stack.

Kim et al. [58] predicts I/O performance using a regression-based approach by integrating system logs from various sources. First, the framework builds a joint database to store logs from different sources and then selects the important features from these logs using different scoring and feature selection algorithms. Once the features are selected, six different regression algorithms are developed, which use these features to predict I/O performance.

2.3.2.3 Replay-based Modeling. Replay-based modeling is another form of modeling and analysis that relies on historical I/O traces or characterization data. These traces, which contain detailed information about an HPC application's

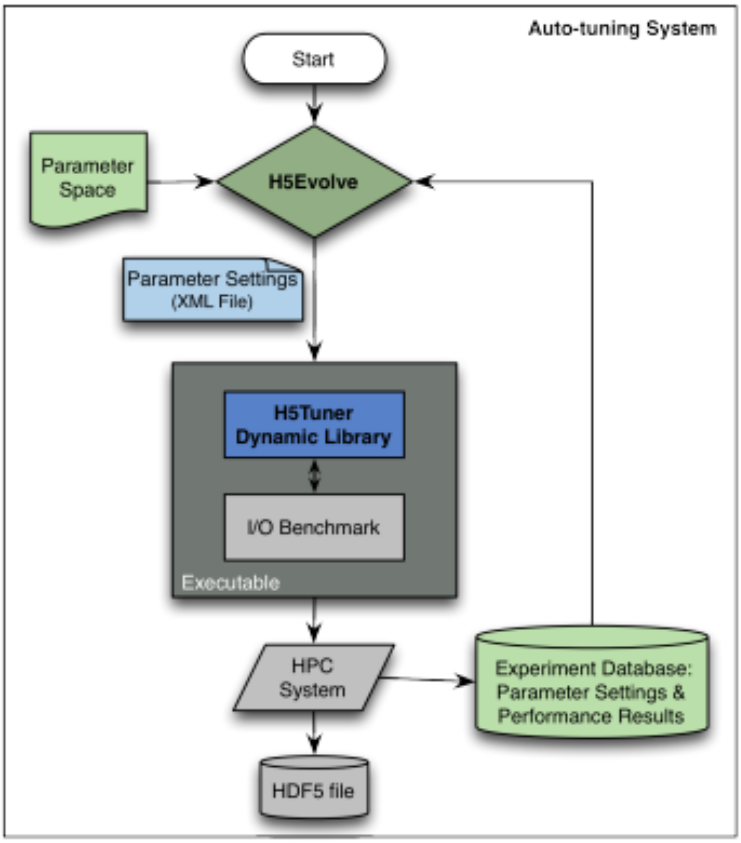


Figure 9. Architecture of the autotuning framework presented in [8]

computation and I/O behavior, can be analyzed to replay the I/O behavior of the original application through I/O workload replication and benchmarks. These benchmarks and workloads can then be further used to predict the I/O performance of the original application. They can also test how the original application will behave in different real-world deployments and scenarios. Apart from this, these replay-based models can also generate workloads for storage systems. There are a variety of tools and research works that study replay-based modeling.

Research/Framework	**Analysis Technique**	**Analysis Goal**
Hao et al [48]	Suffix tree-based compression algorithm	Portable benchmark of the original application to mimic computation, communication, and I/O
ScalaIOExtrap [76]	Trace Extrapolation to generate a single trace	C code of the benchmark for the original I/O application
IOscope [101]	Specific and ready-to-visualize traces of the applications' I/O patterns	Reduce overhead of collecting a large amount of traces
SynchroTrace [104]	Replay mechanism aware of system dependencies	Dependency-aware architecture agnostic traces for multithreaded applications

Table 6. Replay-based modeling tools/research and a brief description of what kind of modeling they do

Hao et al. [48] presents a replay-based modeling framework to generate portable benchmarks for I/O intensive parallel applications automatically. It introduces a trace merging and trace compression algorithm, which it uses to generate benchmarks of the original I/O application, mimicking its computation, communication, and I/O access patterns. These portable benchmarks can also be used to predict the I/O performance of the original application without the extra overhead, as the benchmarks are a scaled-down version of the original I/O application. Fig. 10 shows the overall workflow of this framework. The framework uses a suffix tree-based algorithm to compress the

traces to reduce the redundant data introduced because of the loops. It extracts and compresses these loops to reduce the size of the trace by several orders of magnitude. After compressing the traces, they are merged and are used to generate the C code of the benchmark for the original I/O application.

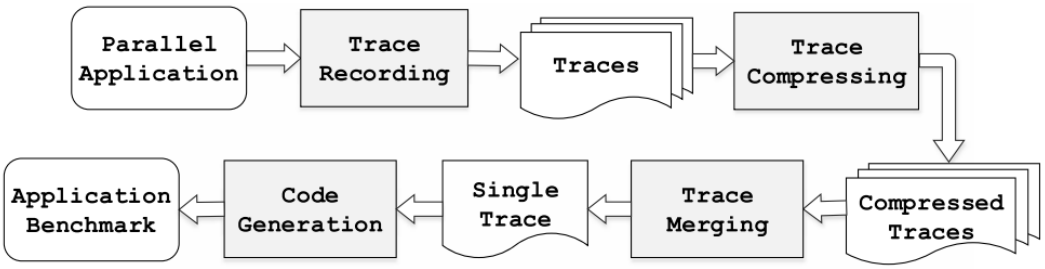


Figure 10. Overall workflow of the proposed framework in [58]

Luo et al. [76] presents an extrapolation and replay-based tool called ScalaIOExtrap and ScalaIOReplay. It builds upon the ScalaIOTrace [128] tool and provides extended functionality for extrapolation and replay-based I/O analysis. It uses ScalaIOTrace to collect and analyze a small number of traces, determining the relationship between the different parameters and the number of ranks. It then uses ScalaIOExtrap to generate a single trace file for an arbitrary number of ranks. The experiments conducted with this tool show that the new I/O trace generated by ScalaIOExtrap retains the trace structure, I/O size, and the number of operations. It also preserves the event ordering and time accuracy in the new trace. This tool enables large-scale parallel I/O evaluation without actually executing the original application.

IOscope [101] is another tracing tool that solves the problem of high overhead incurred by collecting large traces across multiple layers of I/O stack by generating specific and ready-to-visualize traces of the applications' I/O patterns. SynchroTrace [104] is another trace-based simulation tool for multithreaded applications that

generates dependency-aware architecture agnostic traces. It also provides a replay mechanism aware of these dependencies and can simulate synchronization actions to estimate the performance and power of chip multiprocessor systems (CMP).

2.4 Conclusion

Understanding the I/O performance of large-scale workloads is a complex task, given the increasing sophistication of parallel computing hardware and the emergence of workloads in artificial intelligence, machine learning, and big data. To guide researchers and practitioners, this chapter surveys more than 100 research papers on parallel I/O, examining the HPC I/O stack and access patterns to show how requests traverse from high-level abstractions to storage systems, highlighting the resulting complexities. The chapter also provides a structured synthesis of profiling and tracing tools, as well as evaluation techniques ranging from statistical and predictive methods to replay-based modeling, and concludes with recent efforts to bridge the gap between trace collection and optimization without requiring expert intervention. While this demonstrates the breadth of available tools and techniques, it also reveals key limitations: most existing approaches stop short of providing actionable optimization guidance and often introduce significant runtime overhead, limiting their applicability in production HPC environments. These gaps motivate the contributions of the following chapters: Chapter III presents an interactive I/O analysis framework that connects trace data to actionable insights, Chapter IV extends this framework by combining I/O metrics from different sources and linking bottlenecks directly to source code, and Chapter V introduces a lightweight runtime optimization workflow to reduce tuning overhead.

CHAPTER III

BRIDGING THE GAP: AN INTERACTIVE I/O ANALYSIS FRAMEWORK

This chapter contains published material with co-authorship. §3.1, §3.2, §3.3, §3.4 contain text from the published paper in ISC'23. The work was a collaboration between the University of Oregon, the Ohio State University, and Lawrence Berkeley National Laboratory. Co-authors include Dr. Jean Luca Bez, Dr. Suren Byna, and Dr. Boyana Norris. I was the first author and wrote most of the §3.2, §3.3. Jean Luca contributed to §3.1, §3.2.3, §3.4, and also helped with experiments and proofreading. Suren also helped with suggestions and proofreading.

3.1 Introduction

As discussed in Chapter II, the parallel I/O stack on large-scale computing systems exposes numerous tuning parameters and optimization techniques, yet applications frequently experience poor performance due to complex interdependencies across software and hardware layers. Profiling and characterization tools such as Darshan [21] and Recorder [131] are invaluable for diagnosing bottlenecks, but they primarily provide metrics and traces rather than actionable guidance. However, despite the availability of such fine-grained traces, there is a gap between the trace collection, analysis, and tuning steps.

A solution to close this gap requires analyzing the collected metrics and traces, automatically diagnosing the root causes of poor performance, and then providing user recommendations. Towards analyzing the collected metrics, Darshan [21, 31] provides various utilities to summarize statistics. However, their interpretation is left to the user to identify root causes and find solutions. There have been many studies to understand the root causes of performance problems, including IOMiner [134] and Zoom-in I/O analysis [135]. However, these studies and tools are either

application-specific or target general statistics of I/O logs. Existing technologies lack the provision of feedback and recommendation to improve the I/O performance or to increase utilization of I/O system capabilities [22].

To address these three components, i.e., analysis of profiles, diagnosis of root causes, and recommendation of actions, we envision a solution that meets the following criteria based on a visualization approach.

1. Provide interactive visualization based on I/O trace files, allowing users to focus on a subset of MPI processes or zoom in to specific regions of the execution;
2. Display contextual information about I/O calls (e.g., operation type, rank, size, duration, start and end times);
3. Understand how the application issues its I/O requests over time under different facets: operation, request sizes, and spatiality of accesses;
4. Observe transformations as the requests traverse the I/O software stack;
5. Detect and characterize the distinct I/O phases of an application;
6. Understand how the ranks access the file system in I/O operations;
7. Provide an extensible community-driven framework so new visualizations and analysis can be easily integrated;
8. Identify and highlight common root causes of I/O performance problems;
9. Provide a set of actionable items based on the detected I/O bottlenecks.

In this chapter, we propose a novel interactive web-based analysis framework named *"Drishti"* to visualize I/O traces, highlight bottlenecks, and help understand

the I/O behavior of scientific applications. Using *Drishti*, which is based on the nine requirements mentioned above, we aim to fill the gap between the trace collection, analysis, and tuning phases. However, designing this framework has several challenges in analyzing I/O metrics for extracting I/O behavior and illustrating it for users to explore, automatically detecting the I/O performance bottlenecks, and presenting actionable items to users. To tackle these challenges, we devised a solution that contains an interactive I/O trace analysis component for end-users to visually inspect their applications' I/O behavior, focusing on areas of interest and getting a clear picture of common root causes of I/O performance bottlenecks. Based on the automatic detection of I/O performance bottlenecks, our framework maps numerous common and well-known bottlenecks and their solution recommendations that can be implemented by users. Our proof-of-concept uses components and issues that are often investigated by I/O experts when end-users complain about I/O performance. Though some might be obvious to an I/O expert, end-users often face a barrier. *Drishti* seeks to streamline the process and empower the community to solve common I/O performance bottlenecks.

We designed *Drishti* to be scalable through different approaches, and not limited by the number of processes/cores. Our goal in using interactive visualizations is to overcome the limitations of static plots where information and bottlenecks cannot be displayed completely due to pixel limitations. In combination with the I/O analysis and recommendations of triggered issues, one can pinpoint the causes of those issues as they are highlighted in the visualization and zoom into areas of interest. A limiting issue might arise when an application issues millions of small requests using all the ranks for a longer period of time. To tackle this challenge, we provide options to generate multiple time-sliced plots. We also laid out the foundations for a community-

based effort so that additional metrics could be added to *Drishti*, combined into more complex bottleneck detection, and integrated into the interactive visualization component. We demonstrate our framework with multiple case studies and visualize performance bottlenecks and their solutions.

The remainder of this chapter is organized as follows. Our approach to interactively explore I/O behaviors is detailed in §3.2, covering design choices, techniques to detect I/O phases and bottlenecks, and available features. We demonstrate its applicability with case studies in §3.3. We conclude the chapter in §3.4.

3.2 Visualization, Diagnosis, and Recommendations

We have designed and developed *Drishti* based on feedback gathered at two supercomputer facilities, the I/O-research community, and targeted end-users. In the following subsections, we discuss the design choices to support interactive visualizations, I/O behavior analysis, I/O phase detection, and how we efficiently map bottlenecks to a set of actionable items in a user-friendly way. In Fig. 11, we show the various components of *Drishti*.

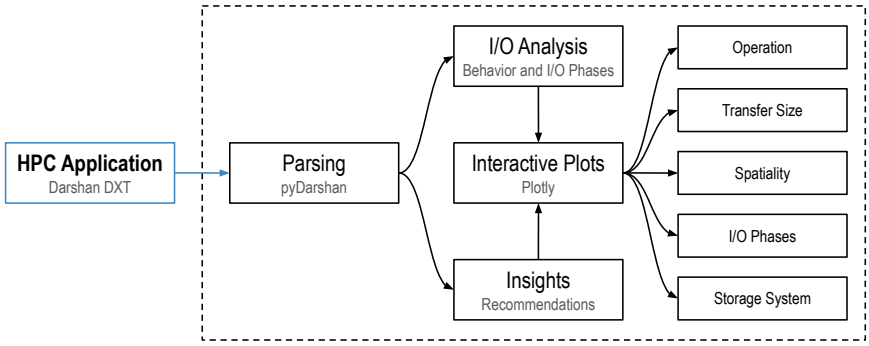


Figure 11. *Drishti* generates meaningful interactive visualizations and a set of recommendations based on the detected I/O bottlenecks using Darshan DXT I/O traces.

3.2.1 Extracting I/O Behavior from Metrics. As discussed in Chapter II, Darshan [21] is a tool deployed on several large-scale computing systems to collect I/O profiling metrics. Darshan collects aggregated statistics with minimal overhead providing a coarse-grain view of application I/O behavior. An extended tracing module of Darshan, DXT [143], can capture fine-grain POSIX and MPI-IO traces. Due to its widespread use, we use Darshan logs as input.

To characterize an application's I/O behavior, we require an efficient way to analyze possibly large traces collected by Darshan DXT logs that are in binary format. Darshan provides a command line solution named `darshan-dxt-parser` as part of the *darshan-util* library to parse DXT traces out of the binary Darshan log files. The parsed data is stored in a pre-defined textual format which could then be transformed into a CSV file to be analyzed. Fig. 12 summarizes the time taken to obtain the required data in such approach. The trace file used in Fig. 12 is an OpenPMD use case with 1024 ranks over 64 nodes. The original trace file was of size 1.9MB and after our transformations, the size was 23.6MB.

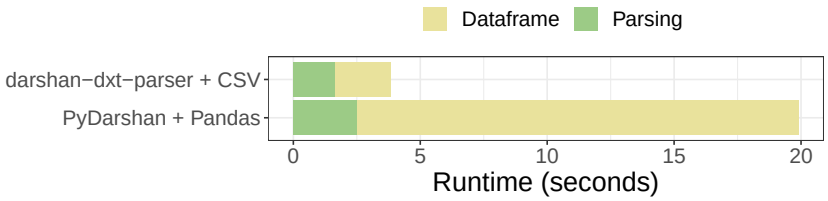


Figure 12. Comparison of methods to extract and combine the I/O behavior metrics from Darshan DXT traces required to pinpoint I/O issues and generate visualizations.

Because of multiple conversions, these additional steps add to the user-perceived time. As an alternative, we have also explored PyDarshan [31], a novel Python package that provides interfaces to binary Darshan log files. With PyDarshan, we get direct access to the parsed DXT trace data in the form of a *pandas* [123] DataFrames.

Fig. 12 compares the performance of both approaches. However, PyDarshan also has shortcomings when the analysis requires an overall view of application behavior. It currently returns a dictionary of DataFrames containing all trace operations issued by each rank. This data structure is not optimal if the visualization requires an overall view of the application behavior, which is the case in *Drishti*. Therefore, an additional step has to be taken to iterate through the dictionary of DataFrames and merge them into a single DataFrame for both analysis and interactive visualization. For the trace in Fig. 12, this additional merging operation represents 87.3% of the time. If PyDarshan can provide direct access to all ranks in form of a single DataFrame, costly data transformations such as the one shown in Fig. 12 can be avoided.

3.2.2 Exploring I/O Behavior Interactively. I/O traces can be large for applications with longer runtimes or even for relatively short applications with a large number of small I/O requests, making analysis and visualization of the behavior difficult. Static plots have space constraints and pixel resolution issues. Thus they often hide the root causes of I/O bottlenecks in plain sight. For instance, when thousands of ranks issue I/O operations concurrently, but some of them suffer interference at the server level, those lines are not visible in a static plot at a regular scale.

Towards developing a modular and extensible framework (criterion 7 in §3.1), we consider two solutions. Our initial prototype to move from a static to interactive and dynamic visualization relied on plots generated in R using *ggplot2*. R is a programming language for statistical computing used in diverse fields such as data mining, bioinformatics, and data analysis. *ggplot2* is an open-source data visualization package for R to declaratively create graphics, based on The Grammar of Graphics [138] schema. A plot generated using this library could be converted into an

interactive visualization by using the open-source *ggplotly* graphing library powered by *Plotly*. *Plotly* is a data visualization library capable of generating dynamic and interactive web-based charts.

However, integrating with the data extraction discussed in §3.2.1 would require the framework to combine features in different languages, compromising modularity, maintainability, and increasing software dependencies, possibly constraining its wide adoption in large-scale facilities. We have opted to rely on PyDarshan to extract the data. Using the open-source Plotly.py Python wrappers would simplify the code without compromising features or usability. Furthermore, it would easily allow I/O data experts to convert their custom visualizations into interactive ones and integrate them into *Drishti*. It also brought the advantage of reducing the total user-perceived time by 84.5% (from avg. of 69.45s to 10.74s), allowing such time to be better spent on detailed analysis of I/O behavior. Fig. 13 summarizes this difference. This is the same OpenPMD use case, with write/read operations, of Fig. 12. Note that Fig. 13 only accounts for I/O phase analysis and plotting. The results on both Fig. 12 and Fig. 13 should be combined for the total runtime, i.e., they depict complementary information. §3.2.3 covers the I/O behavior analysis to pinpoint the root causes of bottlenecks.

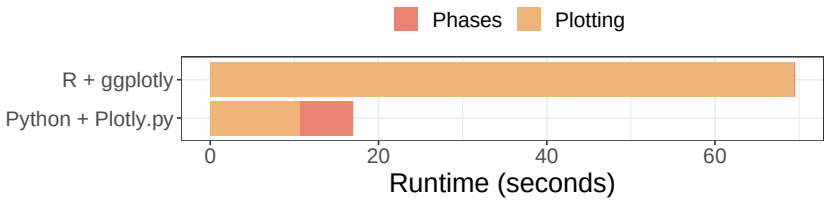
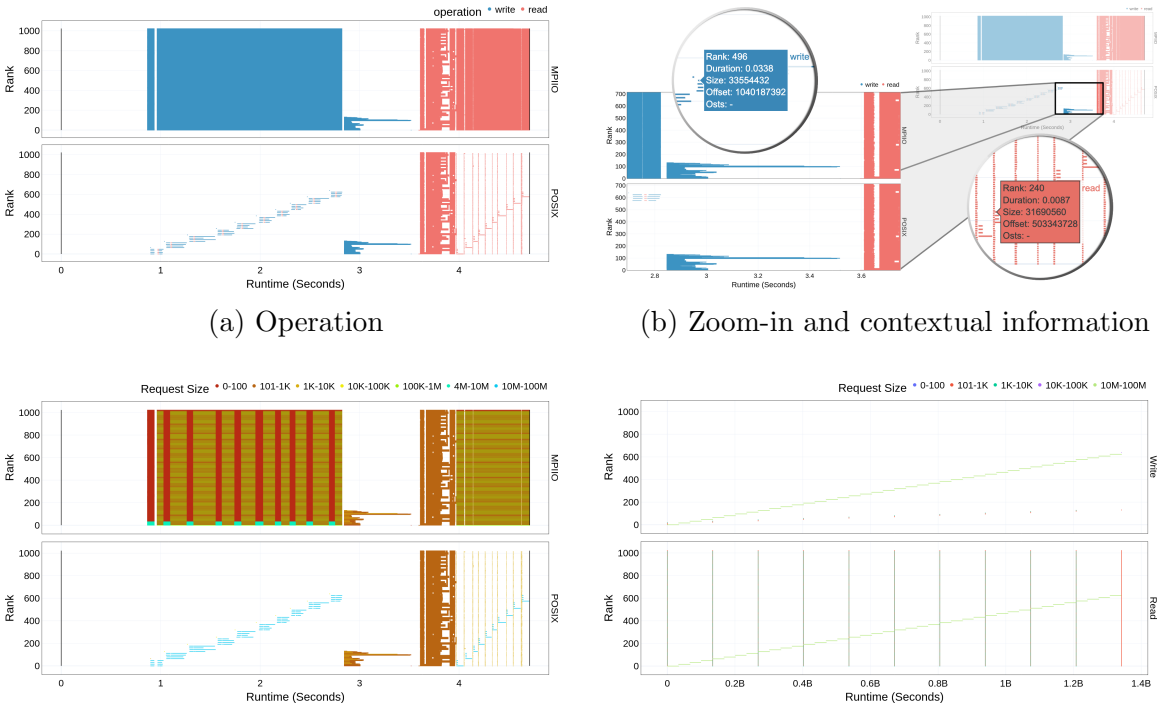


Figure 13. Comparison of solutions to generate the interactive plots and detect I/O phases from Darshan DXT traces. Both approaches use the Plotly.js library under the hood to generate web-based interactive plots.

As scientific applications often handle multiple files during their execution, which overlap in time (e.g., file-per-process or multiple processes to multiple files approaches), *Drishti* should provide a separate visualization for each. Furthermore, those visualizations should shine some light on the application's I/O behavior from multiple perspectives, i.e., criterion 3: operation, data transfer, and spatiality. Fig. 14 shows the reports of particle and mesh-based data from a scientific simulation. Plotly also meets our criteria by allowing a user to dynamically narrow down the plot to cover a time interval of interest or zoom into a subset of ranks to understand the I/O behavior (criterion 1).



(a) Operation

(b) Zoom-in and contextual information

(c) Transfer Size

(d) Spatiality

Figure 14. *Drishti* reports focusing on different facets of the I/O behavior: (a) operations; (b) contextual information regarding the operations; (c) transfer sizes; and (d) spatial locality of the requests into the file. Combined, they provide a clear picture of the I/O access pattern and help identify the root causes of performance problems.

Because of the complexity of the parallel I/O stack, the requests issued by an application are transformed before reaching the file system. Those transformations originate from different mappings between the data model used by an application and its file representation or by the application of I/O optimization techniques such as collective buffering and data-sieving [120] or request scheduling [17, 11, 25]. To shed light on these transformations, *Drishti* depicts every plot using two synchronized facets: the first representing the MPI-IO level, and the second, its translation to POSIX level (criterion 4). For each request, by hovering over the depicted interval, it is possible to inspect additional details such as the operation type, execution time, rank, and transfer size, meeting criterion 2.

When visualizing an application's I/O behavior, we are one step closer to understanding the root causes of any performance bottlenecks, demystifying data transformations, and guiding users to apply the most suitable set of optimization techniques to improve performance. We highlight that there is a lack of a straightforward translation of the I/O bottlenecks into potential tuning options. In this chapter, we seek to close this gap by providing a framework to bring those issues to light, automatically detecting bottlenecks and meaningfully conveying actionable solutions to users.

3.2.3 Automatic Detection of I/O Bottlenecks. Several tools seek to analyze the performance of HPC applications, as discussed in Chapter II. However, few of them focus on I/O and neither provide support for auto-detection of I/O bottlenecks in the application nor provide suggestions on how to fix those. We summarize common root causes of I/O performance bottlenecks in Table 7. Some issues require additional data or a combination of metrics collected from profilers, tracers, and system logs. For instance, Darshan's profiler only keeps track of the

timestamp of the first and last operations to a given file. In contrast, its Extended Tracing module (DXT) tracks what happens in between, such as different behaviors or I/O phases.

Table 7. Root causes of I/O performance bottlenecks

Root Causes	Darshan	DXT	System	Drishti
Too many I/O phases [135]	✓	✓	✗	✓
Stragglers in each I/O phase [119]	✓	✓	✗	✓
Bandwidth limited by a single OST I/O bandwidth [135, 73]	✗	✗	✓	✗
Limited by the small data size [135]	✓	✓	✗	✓
Rank 0 heavy-workload [147]	✓	✓	✗	✓
Unbalanced I/O workload among MPI ranks [135]	✓	✓	✗	✓
Large number of small I/O requests [135]	✓	✓	✗	✓
Unbalanced I/O workload on OSTs [135, 147]	✓	✓	✓	✓
Bad file system weather [135, 72]	✗	✗	✓	✗
Redundant/overlapping I/O accesses [112, 25]	✓	✓	✗	✓
I/O resource contention at OSTs [146, 116]	✗	✗	✓	✗
Heavy metadata load [73]	✓	✗	✗	✓

Drishti seeks to provide interactive web-based visualizations of the tracing data collected by Darshan, but it also provides a framework to detect I/O bottlenecks in the data (from both profiling and tracing metrics) and highlights criterion 8 those on the interactive visualizations along with providing a set of recommendations (criterion 9) to solve the issue. *Drishti* relies on counters available in Darshan profiling logs to detect common bottlenecks and classify the insights into four categories based on the impact of the triggered event and the certainty of the provided recommendation: `HIGH` (high probability of harming I/O performance), `WARN` (detected issues could negatively impact the I/O performance, but metrics might not be sufficient to detect application design, configuration, or execution choices), `OK` (the recommended best practices have been followed), and `INFO` (details relevant information regarding application configuration that could guide tuning solutions). The *insights* module is fully integrated with the parsing and visualization modules of the framework, so the identified issues and actionable items can enrich the reports.

Table 8. Triggers evaluated by *Drishti* for each Darshan log.

Level	Interface	Detected Behavior
HIGH	STDIO	High STDIO usage* (> 10% of total transfer size uses STDIO)
OK	POSIX	High number* of sequential read operations (≥ 80%)
OK	POSIX	High number* of sequential write operations (≥ 80%)
INFO	POSIX	Write operation count intensive* (> 10% more writes than reads)
INFO	POSIX	Read operation count intensive* (> 10% more reads than writes)
INFO	POSIX	Write size intensive* (> 10% more bytes written then read)
INFO	POSIX	Read size intensive* (> 10% more bytes read then written)
WARN	POSIX	Redundant reads
WARN	POSIX	Redundant writes
HIGH	POSIX	High number* of small† reads (> 10% of total reads)
HIGH	POSIX	High number* of small† writes (> 10% of total writes)
HIGH	POSIX	High number* of misaligned memory requests (> 10%)
HIGH	POSIX	High number* of misaligned file requests (> 10%)
HIGH	POSIX	High number* of random read requests (> 20%)
HIGH	POSIX	High number* of random write requests (> 20%)
HIGH	POSIX	High number* of small† reads to shared-files (> 10% of reads)
HIGH	POSIX	High number* of small† writes to shared-files (> 10% of writes)
HIGH	POSIX	High metadata time* (one or more ranks spend > 30 seconds)
HIGH	POSIX	Rank o heavy workload
HIGH	POSIX	Data transfer imbalance between ranks (> 15% difference)
HIGH	POSIX	Stragglers detected among the MPI ranks
HIGH	POSIX	Time imbalance* between ranks (> 15% difference)
WARN	MPI-IO	No MPI-IO calls detected from Darshan logs
HIGH	MPI-IO	Detected MPI-IO but no collective read operation
HIGH	MPI-IO	Detected MPI-IO but no collective write operation
WARN	MPI-IO	Detected MPI-IO but no non-blocking read operations
WARN	MPI-IO	Detected MPI-IO but no non-blocking write operations
OK	MPI-IO	Detected MPI-IO and collective read operations
OK	MPI-IO	Detected MPI-IO and collective write operations
HIGH	MPI-IO	Detected MPI-IO and inter-node aggregators
WARN	MPI-IO	Detected MPI-IO and intra-node aggregators
OK	MPI-IO	Detected MPI-IO and one aggregator per node

* Trigger has a threshold that could be further tunned. Default value in parameters.
† Small requests are consider to be < 1MB.

The interactive visualizations are enhanced using multi-layered plots, with each layer activated according to the detected bottleneck keeping the original behavior in the background (criterion 8). The idea behind highlighting the bottlenecks on the interactive visualizations, apart from classifying the bottlenecks in different categories, is to allow the user to actually visualize where the bottlenecks are in the application. This will allow them to get more detailed information about the bottlenecks and give them more clarity about the application behavior which the textual information alone cannot provide. Furthermore, we complement the interactive visualization with a report based on 32 checks covering common I/O performance pitfalls and good

practices, as summarized in Table 8. We provide the multi-layered plot functionality for the *operation* plot for now. Each layer of the plot shows a different variant of the base graph, for example, one layer can show one of the bottlenecks in the graph, and the other can show the base chart.

3.2.4 Exploring I/O Phases and Bottlenecks. HPC applications tend to present a fairly consistent I/O behavior over time, with a few access patterns repeated multiple times over their execution [71]. Request scheduling [17, 11], auto-tuning [2, 10, 7] and reinforcement-learning [67, 12] techniques to improve I/O performance also rely on this principle to use or find out the best configuration parameters for each workload, allowing the application to fully benefit from it in future iterations or executions. We can define an I/O phase as a continuous amount of time where an application is accessing its data following in a specific way or following one or a combination of access patterns. Nonetheless, factors outside the application's scope could cause an I/O phase to take longer, such as network interference, storage system congestion, or contention, significantly modifying its behavior. Seeking to detect I/O phases, *Drishti* adds an interactive visualization based on DXT trace data. This visualization gives a detailed picture of I/O phases and I/O patterns in the data and is very helpful in extracting information related to bottlenecks such as stragglers, meeting our criterion 5.

Finding the I/O phases from trace data is not trivial due to the sheer amount of data, often representing millions of operations in the order of milliseconds. We use PyRanges [114] to find similar and overlapping behavior between an application's MPI ranks and a threshold value to merge I/O phases closer to each other. PyRanges is a genomics library used for handling genomics intervals. It uses a 2D table to represent the data where each row is an interval (in our case, an operation), and

columns represent chromosomes (i.e., interface and operation), the start and end of an interval (i.e., operation).

While identifying the I/O phases, we keep track of the duration between each I/O phase that represents computation or communication. Once we have the duration of all the intervals between the I/O phases, we calculate the mean and standard deviation of such intervals. A threshold is calculated by summing up the mean and the standard deviation, and it is used to merge I/O phases close to each other into a single I/O phase. We do that because due to the small time scale of the operations, we might end up with a lot of tiny I/O phases that, from the application's perspective, represent a single phase. As of now, the merging threshold cannot be changed dynamically from the visualization interface. Algorithm 1 describes the merging process. We take an I/O phase and check if the difference between the end of the last I/O phase and the start of this I/O phase is less than equal to the threshold value. We keep on merging the I/O phases till they satisfy this condition.

Algorithm 1 Merging I/O phases by a threshold

```
end ← df[end][0]
prev_end ← 0
while i < len(df) do
    if df[start][i] − end <= threshold then
        prev_end ← df[end][i]
    end if
    if df[start][i] − end > threshold OR i = len(df) − 1 then
        chunk_end ← df[prev_index : i].copy()
        end ← df[end][i]
        prev_end ← i
    end if
end while
```

Fig. 15 shows a sample I/O phases visualization, that is fully interactive supporting zoom-in/zoom-out. The phases are generated for MPIIO and POSIX

separately. Hovering over an I/O phase displays the fastest and slowest rank in that phase and their durations.

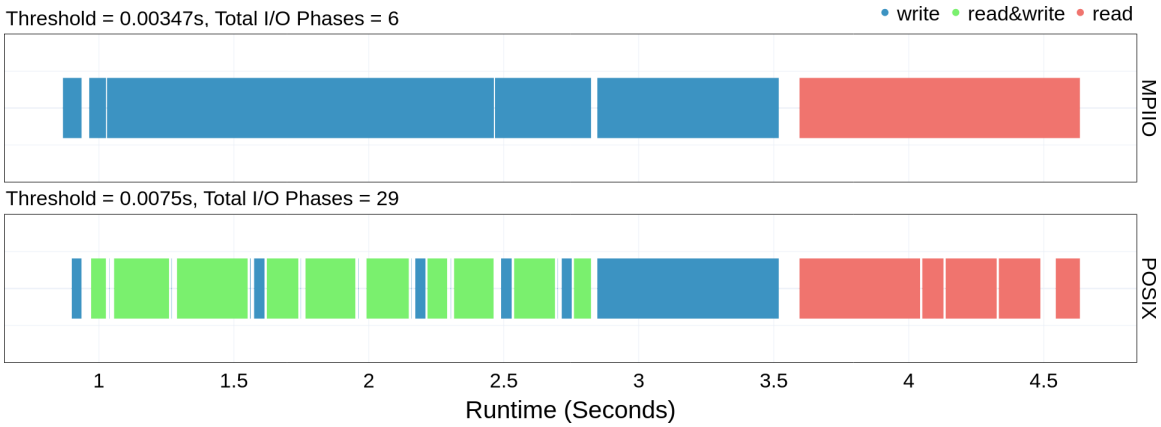


Figure 15. Interactive I/O phases visualization in MPI-I/O and POSIX layers.

Understanding an application's I/O phases allow the detection of additional performance bottlenecks, as detailed by Table 7. To showcase how *Drishti* could be used in this context, we briefly cover synchronous and asynchronous requests, stragglers, and multiple I/O phases.

3.2.4.1 Blocking I/O Accesses. From a scientific application's perspective, I/O operations can be synchronous or asynchronous. Asynchronous I/O is becoming increasingly popular to hide the cost associated with I/O operation and improve overall performance by overlapping computation or communication with I/O operations [117, 89]. Multiple interfaces (e.g., POSIX and MPI-IO) and high-level I/O libraries (e.g., HDF5) provide both blocking and non-blocking I/O calls. For HDF5, the Asynchronous I/O VOL Connector [118] can explore this feature.

If we consider only the profiling data available in Darshan, it only captures the number of non-blocking calls at the MPI-IO level and not when they happened. To provide a detailed and precise suggestion of when asynchronous could benefit the application, we rely on the I/O phases and the intervals between those to provide

such recommendations. We demonstrate a use case with a block-structured adaptive mesh refinement application in §3.3.

3.2.4.2 I/O Stragglers. I/O stragglers in each phase define the critical path impairing performance. *Drishti* has an exclusive visualization to highlight the I/O phases and their stragglers (Fig. 16). We handle each interface separately due to the transformations that happen as requests go down the stack. The dotted lines represent the boundaries of an I/O phase. In each, the fastest and the slowest rank is shown. Combined with contextual information, it is possible to detect slow ranks across the entire execution or storage servers consistently delivering slow performance.

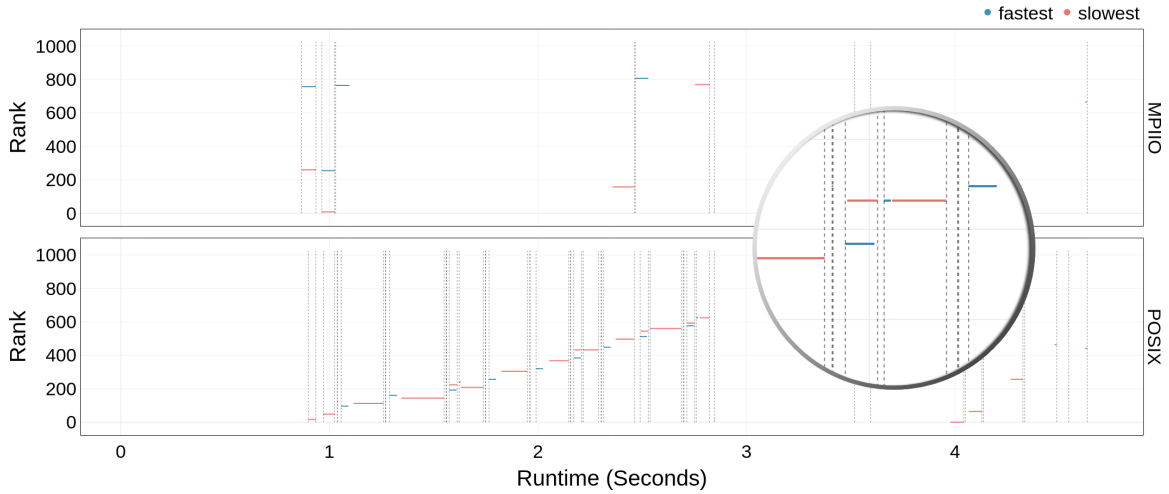


Figure 16. Stragglers are identified in red for each I/O phase.

3.2.5 Towards Exploring File System Usage. Additional logs are required to correctly detect bottlenecks related to unoptimized file system accesses, as detailed in Table 7. Nonetheless, Darshan DXT captures some information that could provide an initial overview of the storage servers' use if the underlying file system is Lustre and that integration is enabled. *Drishti* provides an exclusive visualization to explore the OST usage of the I/O requests, as depicted in Fig. 17. Furthermore, because of file stripping, a request at the MPI-I/O level might be broken down and require access to multiple storage devices to be completed, which explains why the

information at both levels is not the same. *Drishti* can also depict the data transfer sizes (writes and reads) for each OST at both the MPI-I/O and POSIX levels.

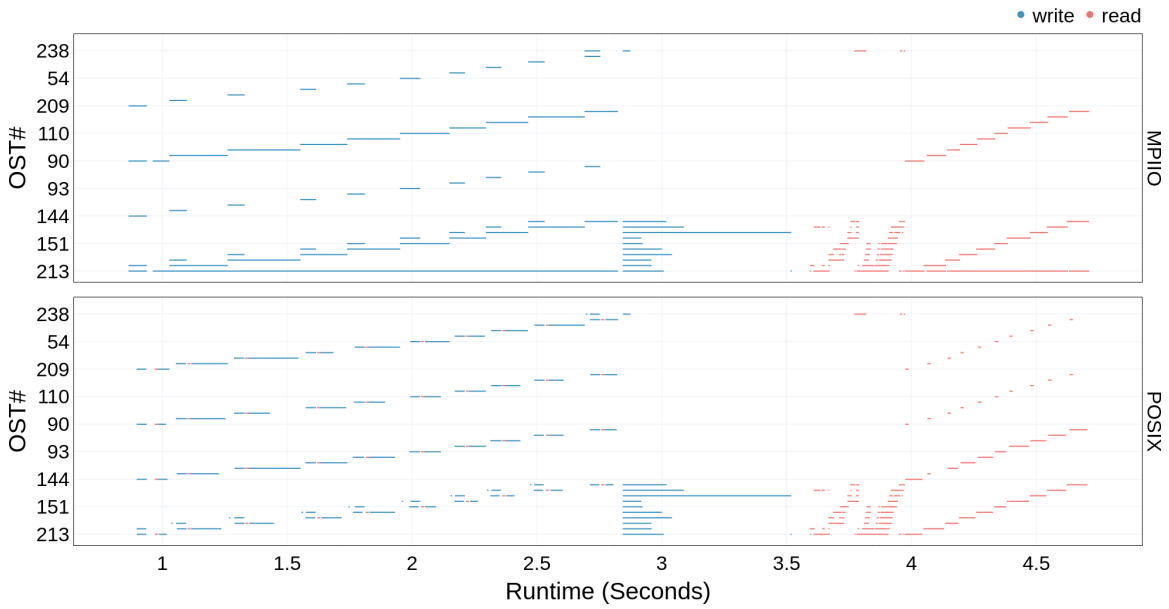


Figure 17. Lustre data storage (OST) access over time.

3.3 Evaluation

We selected the OpenPMD (§3.3.2) and AMReX (§3.3.3) use cases from distinct science domains to demonstrate *Drishti*'s value to end-users, developers, and system administrators. Both came from interactions with their core developers about concerns related to poor I/O performance. Existing solutions previously tried did not uncover all the root causes of performance inefficiencies. Experiments were conducted in two production supercomputing systems: Cori at the National Energy Research Scientific Computing Center (NERSC) and Summit at the Oak Ridge Leadership Computing Facility (OLCF). We have also probed the I/O research community and targeted end-users to gather feedback on the tool's features and helpfulness. For instance, highlighting bottlenecks uncovered by the heuristic analysis, indexing, and filtering the generated visualizations based on the file are some enhancements added

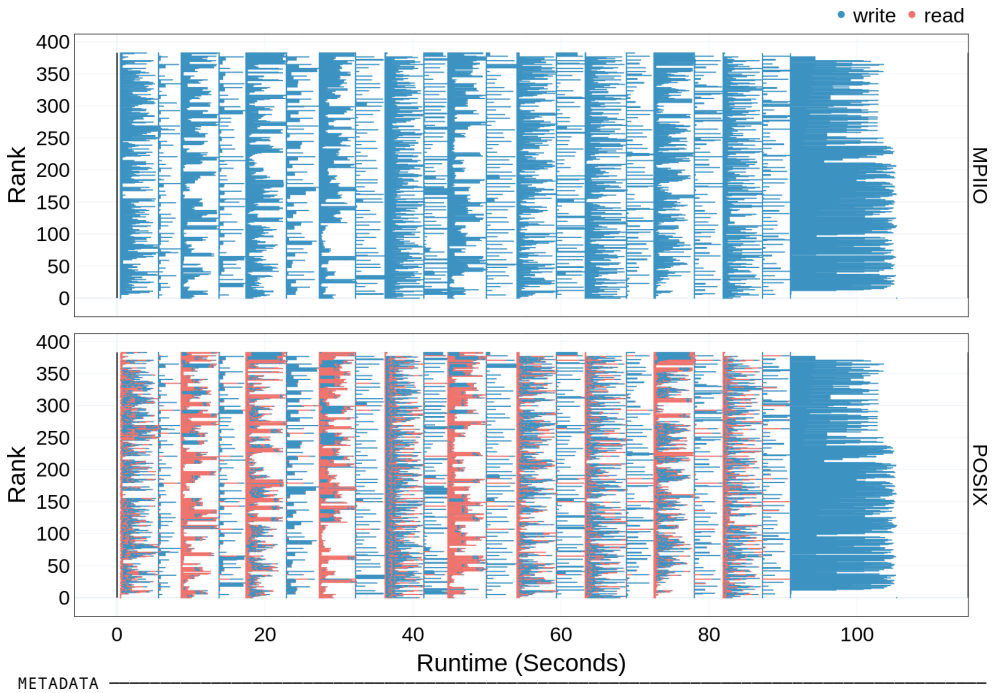
from community-driven feedback. User-interface presentation of the contextual data was also shaped based on such evaluation.

3.3.1 I/O Systems in NERSC and OLCF. Cori is a Cray XC40 supercomputer at NERSC. It has 2,388 Intel Xeon Haswell, and 9,688 Intel Xeon Phi Knight's Landing (KNL) compute nodes. All compute nodes are connected to a ≈ 30 PB Lustre parallel file system with a peak I/O bandwidth of 744 GB/s. Cori's PFS is comprised of 244 Object Storage Servers.

Summit is a 4,608 compute nodes IBM supercomputer at OLCF. Summit is connected to a center-wide 250 PB Spectrum Scale (GPFS) file system, with a peak bandwidth of 2.5 TB/s. It has 154 Network Shared Disk servers, each managing one GPFS Native RAID serving as data and metadata server.

3.3.2 I/O Bottlenecks in OpenPMD. Open Standard for Particle-Mesh Data Files (OpenPMD) [51] is an open meta-data schema targeting particle and mesh data in scientific simulations and experiments. Its library [60] provides back-end support for multiple file formats such as HDF5 [40], ADIOS [68], and JSON [94]. In the context of this experiment, we focus on the HDF5 format to store the 3D mashes $[65536 \times 256 \times 256]$, represented as grids of $[64 \times 32 \times 32]$ composed by $[64 \times 32 \times 32]$ mini blocks. The kernel runs for 10 iteration steps writing after each one. Figure 18 depicts a baseline execution of OpenPMD in the Summit supercomputer, with 64 compute nodes, 6 ranks per node, and 384 processes, prior to applying any I/O optimizations alongside the triggered issues. For this scenario, OpenPMD takes on average 110.6 seconds (avg. of 5 runs).

Based on the initial visualization and the provided report (Fig. 18), it becomes evident that the application I/O calls are not using MPI-IO's collective buffering tuning option. Furthermore, the majority of the write and read requests are small



METADATA

▸ Application is write operation intensive (60.83% writes vs. 39.17% reads)
▸ Application is write size intensive (64.15% write vs. 35.85% read)
▸ Application issues a high number (100.00%) of misaligned file requests
↪ **Recommendations:**
↪ Consider aligning the requests to the file system block boundaries

OPERATIONS

▸ Application issues a high number (275840) of small read requests (i.e., < 1MB) which represents 100.00% of all read/write requests
↪ 275840 (100.00%) small read requests are to "8a_parallel_3Db_0000001.h5"
↪ **Recommendations:**
↪ Consider buffering read operations into larger more contiguous ones
↪ Since the appplication already uses MPI-IO, consider using collective I/O calls (e.g. MPI_File_read_all() or MPI_File_read_at_all()) to aggregate requests into larger ones
▸ Application issues a high number (427386) of small write requests (i.e., < 1MB) which represents 99.75% of all read/write requests
↪ 275840 (64.38%) small write requests are to "8a_parallel_3Db_0000001.h5"
↪ **Recommendations:**
↪ Consider buffering write operations into larger more contiguous ones
↪ Since the application already uses MPI-IO, consider using collective I/O calls (e.g. MPI_File_write_all() or MPI_File_write_at_all()) to aggregate requests into larger ones
▸ Application mostly uses consecutive (97.67%) and sequential (2.16%) read requests
▸ Application mostly uses consecutive (97.85%) and sequential (1.17%) write requests
▸ Detected read imbalance when accessing 1 individual files.
↪ Load imbalance of 55.23% detected while accessing "8a_parallel_3Db_0000001.h5"
↪ **Recommendations:**
↪ Consider better balancing the data transfer between the application ranks
↪ Consider tuning the stripe size and count to better distribute the data
↪ If the application uses netCDF and HDF5 double-check the need to set NO_FILL values
↪ If rank 0 is the only one opening the file, consider using MPI-IO collectives
▸ Application uses MPI-IO and write data using 7680 (92.50%) collective operations
▸ Application could benefit from non-blocking (asynchronous) reads
↪ **Recommendations:**
↪ Since you use HDF5, consider using the ASYNC I/O VOL connector (https://github.com/hpc-io/vol-async)
↪ Since you use MPI-IO, consider non-blocking/asynchronous I/O operations
▸ Application could benefit from non-blocking (asynchronous) writes
↪ **Recommendations:**
↪ Since you use HDF5, consider using the ASYNC I/O VOL connector (https://github.com/hpc-io/vol-async)
↪ Since you use MPI-IO, consider non-blocking/asynchronous I/O operations

Figure 18. Interactive visualization and recommendations report generated by *Drishti* for the OpenPMD baseline execution in Summit.

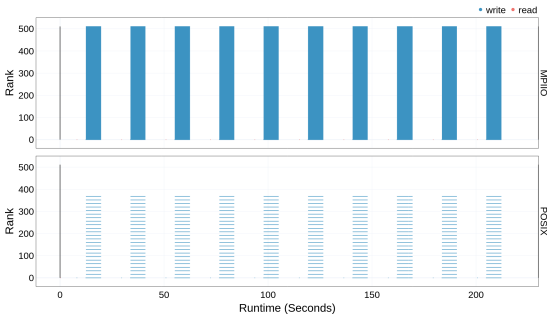
(< 1MB), which is known to have a significant impact on I/O performance [135]. Moreover, *Drishti* has detected an imbalance when accessing the data. This is further highlighted when the user selects that issue in the interactive web-based visualization.

Nonetheless, after careful investigation, we confirmed that the application and the HDF5 library supposedly used collective I/O calls, though the visualization depicted something entirely different. *Drishti* aided in discovering an issue introduced in HDF5 1.10.5 that caused collective operations to be instead issued as independent by the library. Once that was fixed, we noticed that the application did not use collective metadata operations. Furthermore, *Drishti* reported misaligned accesses which pointed us toward tuning the MPI-I/O ROMIO collective buffering and data sieving sizes to match Alpine's 16MB striping configuration and the number of aggregators. Following the recommendations provided by *Drishti*, the runtime dropped to 16.1 seconds, a 6.8× speedup from the baseline execution.

3.3.3 Improving AMReX with Asynchronous I/O. AMReX [149] is a C++ framework developed in the context of the DOE's Exascale Computing Project (ECP). It uses highly parallel adaptive mesh refinement (AMR) algorithms to solve partial differential equations on block-structured meshes. AMReX-based applications span different areas such as astrophysics, atmospheric modeling, combustion, cosmology, multi-phase flow, and particle accelerators. We ran AMReX with 512 ranks over 32 nodes in Cori 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. Table 9 (left) shows the interactive baseline execution and the report generated by *Drishti*.

Table 9. *Drishti* report generated for the AMReX in Cori.

Baseline AMReX



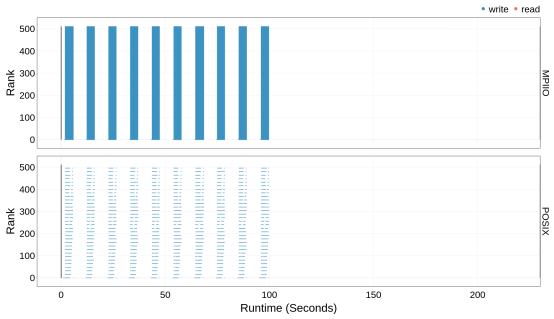

```
METADATA

▸ Application is write operation intensive (99.98% writes vs. 0.02% reads)
▸ Application is write size intensive (100.00% write vs. 0.00% read)

OPERATIONS

▸ Application issues a high number (491640) of small write requests (i.e., < 1MB) which
represents 99.99% of all read/write requests
   ↪ 98328 (20.00%) small write requests are to "plt00001.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00002.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00005.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00009.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00000.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00004.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00003.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00006.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00007.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00008.h5"
   ↪ Recommendations:
   ↪ Consider buffering write operations into larger more contiguous ones
   ↪ Since the application already uses MPI-IO, consider using collective I/O calls (e.g.
MPI_File_write_all() or MPI_File_write_at_all()) to aggregate requests into larger ones
▸ Application mostly uses consecutive (25.41%) and sequential (32.79%) read requests
▸ Application mostly uses consecutive (0.01%) and sequential (99.98%) write requests
▸ Application issues a high number (491640) of small write requests to a shared file (i.e., <
1MB) which represents 99.99% of all shared file write requests
   ↪ 49164 (10.00%) small writes requests are to "plt00001.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00002.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00005.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00009.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00000.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00004.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00003.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00006.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00007.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00008.h5"
   ↪ Recommendations:
   ↪ Consider coalescing write requests into larger more contiguous ones using MPI-IO
collective operations
▸ Application uses MPI-IO and write data using 15360 (99.81%) collective operations
▸ Application could benefit from non-blocking (asynchronous) reads
   ↪ Recommendations:
   ↪ Since you use HDF5, consider using the ASYNC I/O VOL connector
(https://github.com/hpc-io/vol-async)
   ↪ Since you use MPI-IO, consider non-blocking/asynchronous I/O operations
▸ Application could benefit from non-blocking (asynchronous) writes
   ↪ Recommendations:
   ↪ Since you use HDF5, consider using the ASYNC I/O VOL connector
(https://github.com/hpc-io/vol-async)
   ↪ Since you use MPI-IO, consider non-blocking/asynchronous I/O operations
```

Optimized AMReX




```
METADATA

▸ Application is write operation intensive (99.61% writes vs. 0.39% reads)
▸ Application is write size intensive (100.00% write vs. 0.00% read)

OPERATIONS

▸ Application mostly uses consecutive (24.79%) and sequential (33.06%) read requests
▸ Application mostly uses consecutive (0.16%) and sequential (99.64%) write requests
▸ Application uses MPI-IO and write data using 15360 (99.81%) collective operations
```

From the provided recommendations, since AMReX uses the high-level HDF5 library, we have added the asynchronous I/O VOL Connector [118] so operations are non-blocking and we could hide some of the time spent in I/O while the application continues its computation. Furthermore, as *Drishti* looks at the ratio of operations to trigger some insights, for this particular case, we can verify that the majority of write requests are small ($<$ 1MB) for all 10 plot files. To increase those requests, we have set the stripe size to 16MB. Table 9 (right) shows the optimized version with a total speedup of 2.1$\times$ (from 211 to 100 seconds).

As demonstrated by design choices and these two use cases, *Drishti* meets all the initial criteria (defined in §2.1) we set to close the gap between analyzing the collected I/O metrics and traces, automatically diagnosing the root causes of poor performance, and then providing users with a set of actionable suggestions. The designed solution provides a framework that can further be extended and refined by the community to encompass additional triggers, interactive visualizations, and recommendations. We have also conducted a similar analysis for h5bench [16] and the end-to-end (E2E)[75] domain decomposition I/O kernel. These are available in our companion repository.

3.4 Conclusion

Pinpointing the root causes of I/O inefficiencies in scientific applications requires detailed metrics and an understanding of the HPC I/O stack. The existing tools lack detecting I/O performance bottlenecks and providing a set of actionable items to guide users to solve the bottlenecks considering each application's unique characteristics and workload. In this chapter, we design a framework to face the challenges in analyzing I/O metrics: extracting I/O behavior and illustrating it for users to explore interactively, detecting I/O bottlenecks automatically, and presenting a set of recommendations to avoid them. *Drishti*, an interactive web-based analysis framework, seeks to close this gap between trace collection, analysis, and tuning. Our framework relies on the automatic detection of common root causes of I/O performance inefficiencies by mapping raw metrics into common problems and recommendations that can be implemented by users. We have demonstrated its applicability and benefits with the OpenPMD and AMReX scientific applications to improve runtime.

While *Drishti* narrows the disconnect between profiling and optimization, it does not yet capture the precise source-level origins of inefficiencies or leverage

additional system-level data sources. The next chapter addresses these limitations by broadening the framework with support for multiple log sources, such as Recorder traces, and by integrating source code analysis to map performance bottlenecks directly to application code. This extension enhances both the diagnostic depth and the actionability of the framework, enabling more precise and effective optimizations.

CHAPTER IV

CROSS-LAYER I/O PROFILING AND SOURCE CODE ANALYSIS FOR BOTTLENECK DIAGNOSIS

This chapter contains published material with co-authorship. §4.1, §4.2, §4.3, §4.4, §4.5, §4.6 contain text from the published paper in IPDPS'24. The work was a collaboration between the University of Oregon, the Ohio State University, and Lawrence Berkeley National Laboratory. Co-authors include Dr. Jean Luca Bez, Yankun Xia, and Dr. Suren Byna. I was the joint first author along with Jean Luca and wrote most of §4.3 and §4.5. Jean Luca contributed to §4.1, §4.4, §4.6 while also contributing to §4.5. Yankun contributed to §4.2. Suren helped with suggestions and proofreading.

4.1 Introduction

As previously discussed in Chapter III, HPC systems provide a multi-layered I/O software stack to support serial and parallel data access from scientific applications. 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 PFS [15]. 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, 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 [54, 55, 93, 98], 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 [129, 21, 63, 92] 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 [14, 20]. To complicate matters further, optimization techniques such as collective buffering and data sieving [120], which have been proposed decades ago and demonstrate performance gains are not yet adopted in cases it should have been. Automatic tuning mechanisms [2, 10, 7] 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.

This dissertation introduced *Drishti* in Chapter III which attempts to fill the gap between the collected metrics and their interpretations and translation into

optimizations through interactive visualizations. Although *Drishti* takes one step in the right direction, it 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 chapter, 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 chapter is organized as follows. In §4.2, we examine different sources of I/O-related metrics, their opportunities, and limitations. §4.3 discusses the design choices to drill down to the root causes in the source code. §4.4 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 §4.5. We conclude the chapter in §4.6 and discuss future R&D.

4.2 Data Sources and Limitation

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. In depth details can be found in Chapter II.

4.2.1 Darshan. As discussed previously in Chapter II, Darshan [21] 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. 19 depicts the extensible format used by Darshan to store these profiling metrics.

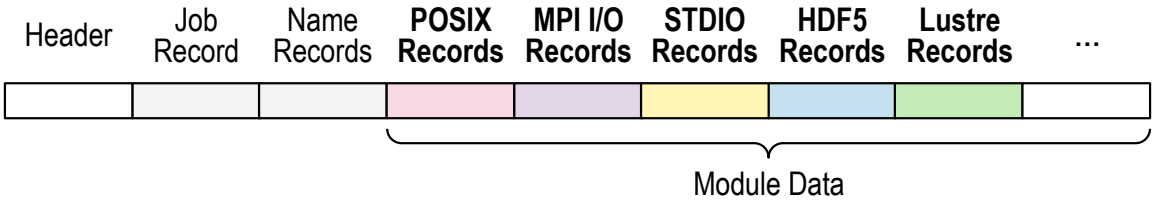


Figure 19. Darshan format to store I/O metrics from multiple modules/sources.

4.2.2 Darshan DXT. Darshan eXtended Tracing (DXT) [143] 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.

4.2.3 Recorder. Recorder [131] 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 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.

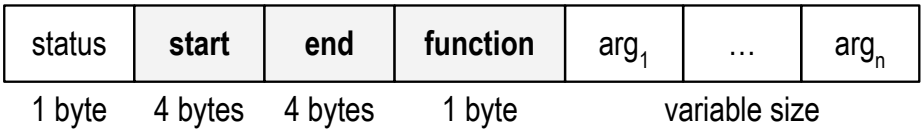


Figure 20. Recorder format representation to store I/O traces.

Fig. 20 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.

4.2.4 High-Level Libraries. Darshan and Recorder have partial support for collecting high-level library I/O counters. 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 4.5.

4.2.5 Parallel File Systems (PFS). In HPC systems, applications read and write data to shared 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 [52, 44] and IBM Spectrum Scale (previously known as GPFS) [106] 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* [95] and Lustre Monitoring Tool (LMT) [42]. 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.

4.3 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.

4.3.1 Unveiling the Source-Code. A backtrace is a list of function calls that are currently active in a program. Though external debuggers are often 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[1].

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

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

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. 21 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. 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.

To get the line number from the addresses, one needs access to DWARF (Debugging With Attributed Record Formats) [29], used by many compilers and debuggers to support source-level debugging. *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,

```
/darshan/lib/libdarshan.so(dxt_posix_write+0x118) [0
   x7f4afdbabc58]
/darshan/lib/libdarshan.so(pwrite+0x19d) [0x7f4afdb9761d]
/opt/cray/pe/lib64/libmpi_gnu_91.so.12(+0x26007f3) [0
   x7f4afcae37f3]
/opt/cray/pe/lib64/libmpi_gnu_91.so.12(+0x2605cec) [0
   x7f4afcae8cec]
/opt/cray/pe/lib64/libmpi_gnu_91.so.12(+0x25d60d9) [0
   x7f4afcab90d9]
/opt/cray/pe/lib64/libmpi_gnu_91.so.12(PMPI_File_write_at_all
   +0x21)
[0x7f4afcabaa11]
/darshan/lib/libdarshan.so(MPI_File_write_at_all+0xbb)
[0x7f4afdbaff1b]
/hdf5/lib/libhdf5.so.310(+0x38913c) [0x7f4afd65e13c]
/hdf5/lib/libhdf5.so.310(H5FD_write+0x68) [0x7f4afd41cda8]
/hdf5/lib/libhdf5.so.310(H5F__accum_write+0x194) [0
   x7f4afd3f6794]
/hdf5/lib/libhdf5.so.310(H5PB_write+0x6cb) [0x7f4afd50f3fb]
/hdf5/lib/libhdf5.so.310(H5F_shared_block_write+0x30)
[0x7f4afd401ce0]
/hdf5/lib/libhdf5.so.310(H5D__mpio_select_write+0x22)
[0x7f4afd65c402]
/hdf5/lib/libhdf5.so.310(+0x37a8d3) [0x7f4afd64f8d3]
/hdf5/lib/libhdf5.so.310(+0x37aaa5) [0x7f4afd64faa5]
/hdf5/lib/libhdf5.so.310(+0x3854e2) [0x7f4afd65a4e2]
/hdf5/lib/libhdf5.so.310(H5D__collective_write+0x9) [0
   x7f4afd65c4c9]
/hdf5/lib/libhdf5.so.310(H5D__write+0x166) [0x7f4afd3c0586]
/hdf5/lib/libhdf5.so.310(H5VL__native_dataset_write+0x8f)
[0x7f4afd61dfdf]
/hdf5/lib/libhdf5.so.310(H5VL_dataset_write_direct+0x81)
[0x7f4afd6085d1]
/hdf5/lib/libhdf5.so.310(+0xb5ef0) [0x7f4afd38aef0]
/hdf5/lib/libhdf5.so.310(H5Dwrite+0x9f) [0x7f4afd38e20f]
/darshan/lib/libdarshan.so(H5Dwrite+0xd7) [0x7f4afdbbc757]
/h5bench/bin/h5bench_e3sm() [0x6c986b]
/h5bench/bin/h5bench_e3sm() [0x457c4b]
/h5bench/bin/h5bench_e3sm() [0x454e87]
/h5bench/bin/h5bench_e3sm() [0x452947]
/h5bench/bin/h5bench_e3sm() [0x451f1c]
/lib64/libc.so.6(__libc_start_main+0xef) [0x7f4af9fba24d]
/h5bench/bin/h5bench_e3sm() [0x4525da]
```

Figure 21. Sample backtrace from an HDF5 application monitored with Darshan.

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. 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. 22, which shows the mapping of the addresses reported for the `h5bench_e3sm` binary (in purple) in Fig. 21.

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

Figure 22. Mapping of stack addresses to source-code lines.

4.3.1.1 ***Feasibility.***

We prototyped a simple solution to compare *addr2line* and *pyelftools* on the h5bench [16] 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. 23.

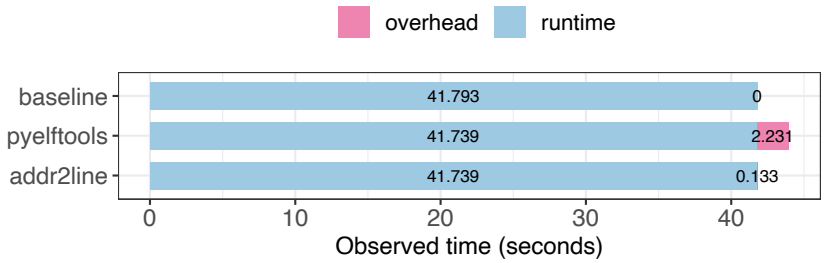


Figure 23. Overhead for getting the line number from the addresses returned by `backtrace()` using different libraries.

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. 24 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 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.

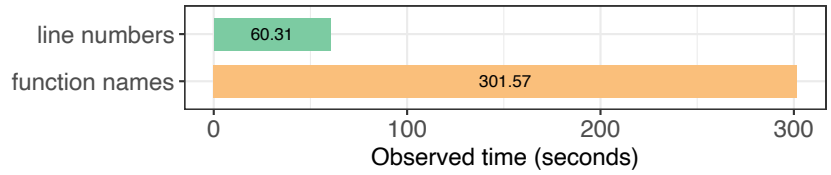


Figure 24. Time taken to get the line numbers vs. the function names using *pyelftools* for all addresses returned by `backtrace()` in an execution with the AMReX [148] I/O kernel (1 compute node and 8 ranks).

4.3.1.2 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. 25. 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. 21 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. 22 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 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. 25 illustrates the complete framework.

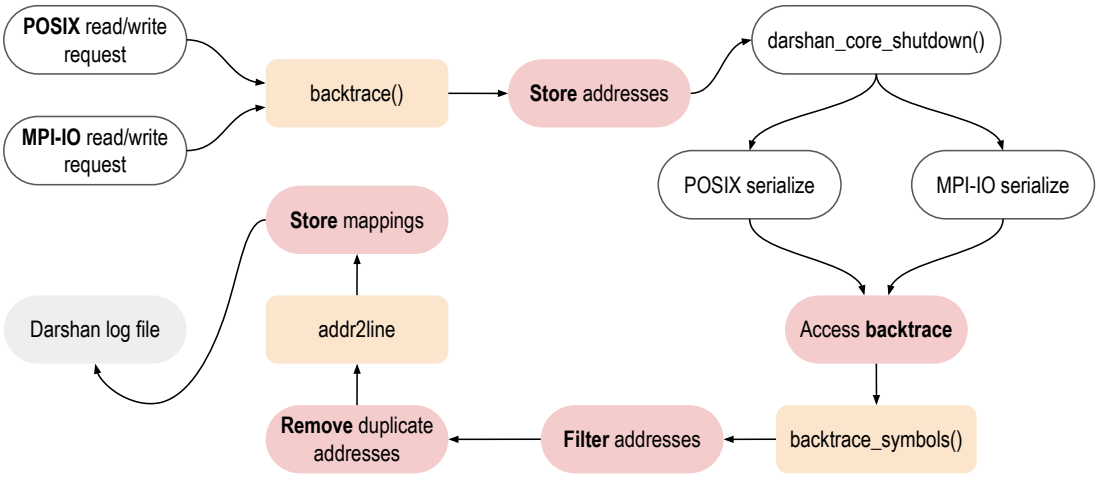


Figure 25. Framework to capture source code information in Darshan to enhance I/O analysis and recommendations in *Drishti*.

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.

4.3.1.3 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()`[2] instead of using the standard `system()` call to optimize this further.

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

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 4.3.1.2. 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 4.5.

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.

4.4 Building up to High Level Libraries

As discussed in Section 4.2.4, 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 [18] 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 chapter 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

Table 10. 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	✗	✗

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 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 10 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 10) 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).

4.5 Evaluation

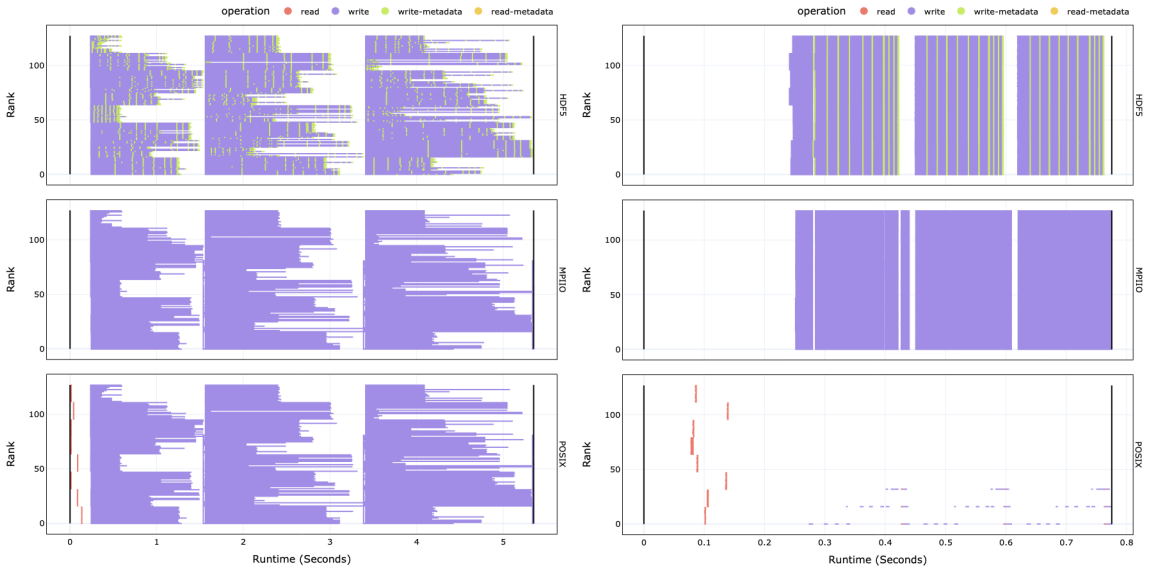
In this section, we demonstrate the potential of our cross-layer *Drishti* exploration solution with three use cases from distinct science domains, by harnessing different I/O metrics.

4.5.1 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 [51] format(s). While plotfiles are AMReX's native data format, openPMD is implemented in popular community formats such as ADIOS [68] and HDF5 [40].

Thus, the optimization of data access using HDF5 plays a crucial role in improving the application's data access performance.

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. 26 illustrates our investigation. The baseline (Fig. 26a) 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 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



(a) Baseline execution (b) Optimized execution

Figure 26. Interactive web-based cross-layer visualization of I/O requests for a WarpX (OpenPMD) execution using multi-source metrics (*Drishti* VOL connector traces, DXT, and Darshan). Optimized execution time has a speedup of 6.9× compared to the baseline.

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. 27. 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

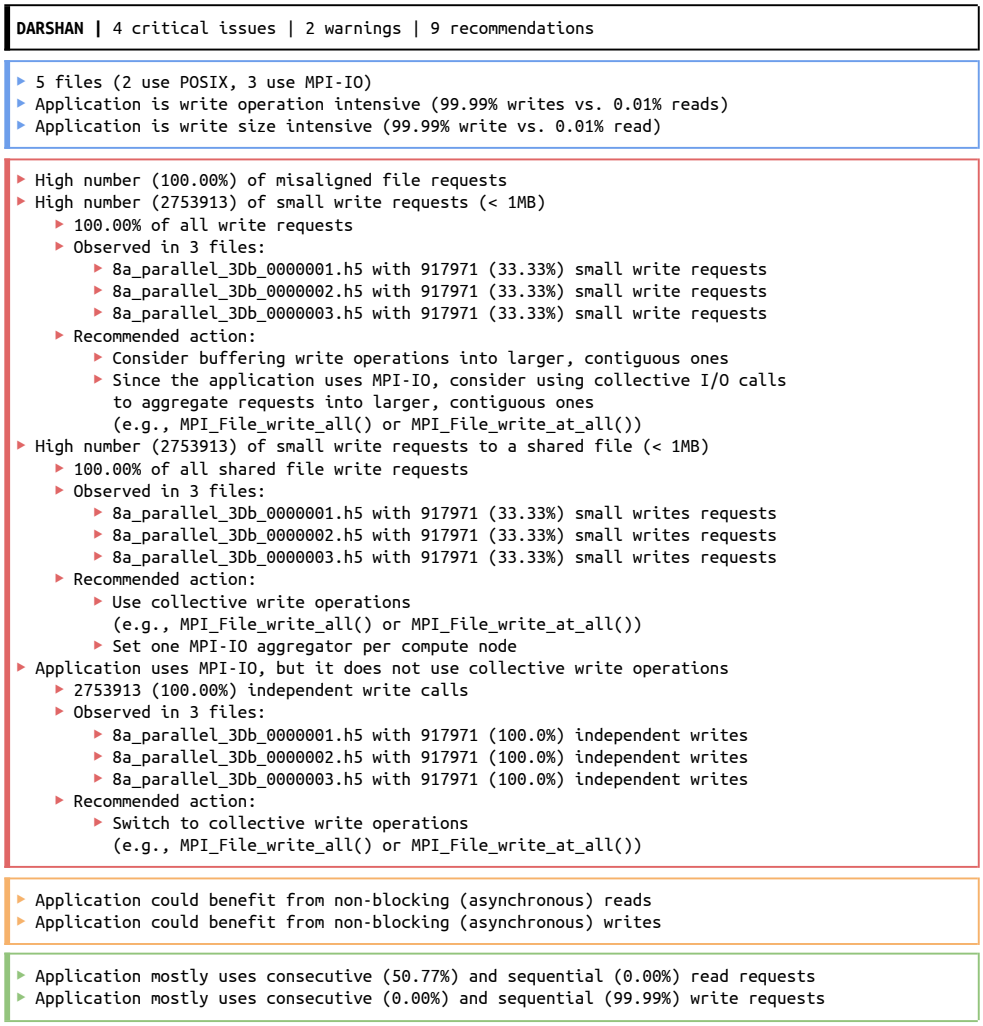


Figure 27. Cross-layer I/O report and insights for openPMD extracted from *Drishti* VOL connector combined with Darshan metrics and DXT traces.

26b 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× speedup.

Table 11. 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

We also investigate the non-negligible overhead of collecting data from multiple layers of the I/O stack. Table 11 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 is increased since every I/O call is intercepted by Darshan. Furthermore, as demonstrated by Fig. Fig. 27 and Fig. 26a, 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.

Apart from getting the overhead for the cross-layer analysis, we also measure the overhead for the source code analysis. For this experiment, we used E3SM-IO with 8

nodes and 512 ranks. Table 12 summarizes these findings. As expected, there is an increase in time as compared to the original run when doing the source code analysis. The overhead is mainly because of the external system call which we have to make for *addr2line*. We acknowledge that there is some overhead for performing the source code analysis but that is only going to be present in the initial exploratory runs of the program when the user is trying to understand the I/O behavior of the application and trying to tune it. Once the valuable insights are gained from *Drishti*, users will not have to incur any 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 as compared to when both Darshan and DXT trace collection are enabled.

4.5.2 AMReX. AMReX [149] 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.

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. 28. 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

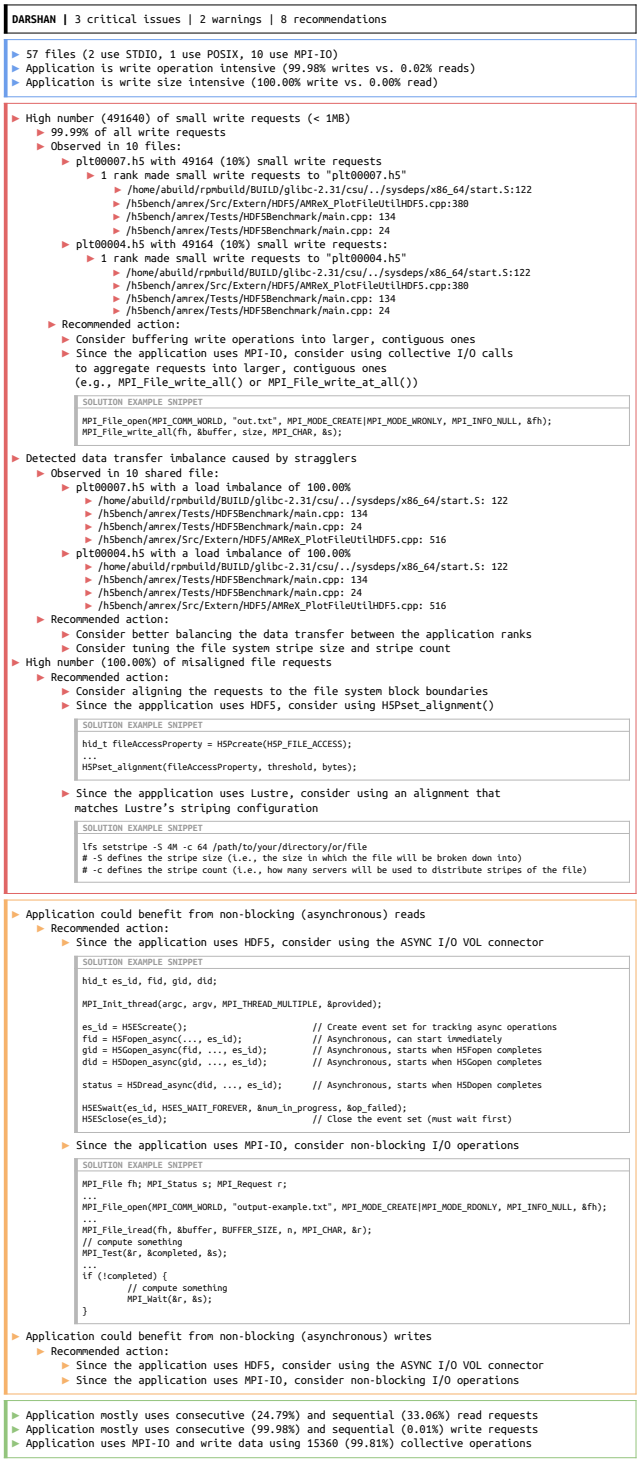


Figure 28. Cross-layer I/O report and insights for the baseline (run-as-is) execution of AMReX in Perlmutter (NERSC) based on Darshan metrics/traces. This sample was generated with the verbose mode which includes source-code and configuration snippets.

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.

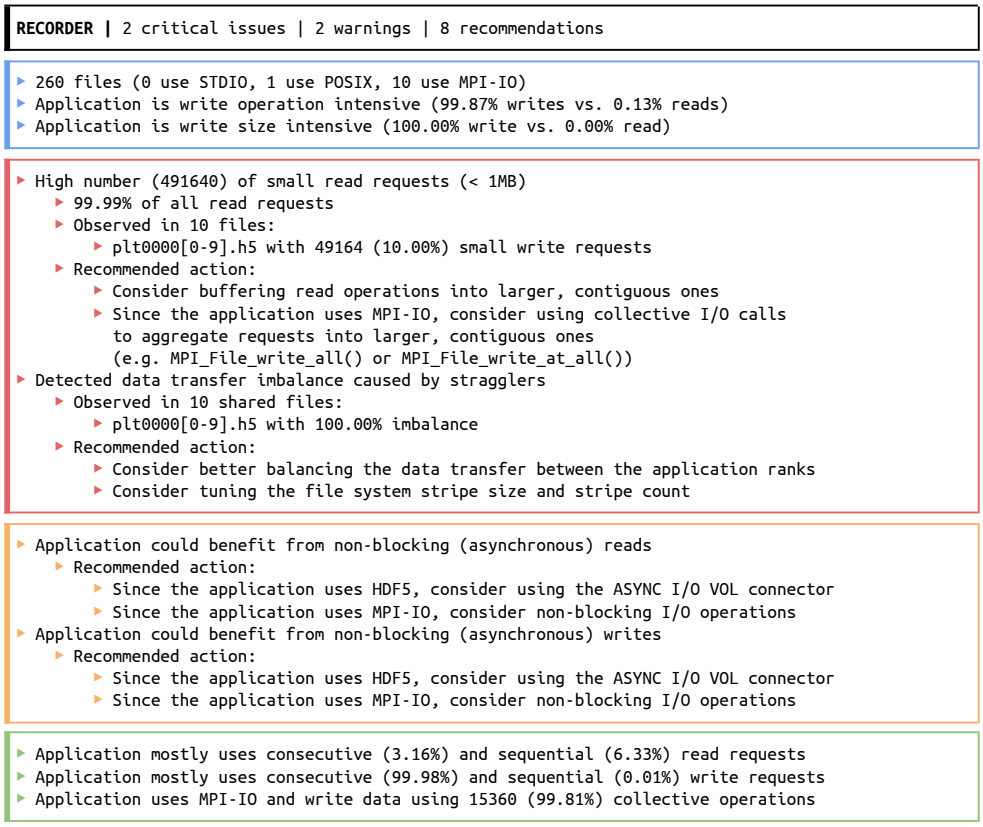


Figure 29. Cross-layer I/O report and insights for the baseline (run-as-is) execution of AMReX in Perlmutter (NERSC) based on Recorder metrics/traces.

Fig. 29 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 4.2), 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).

4.5.3 Energy Exascale Earth System Model (E3SM). E3SM-IO is the parallel I/O kernel from the Energy Exascale Earth System Model (E3SM) [38, 46] climate simulation model. It makes use of Parallel I/O Library (PIO) [50] which is built on top of PnetCDF [66]. For the evaluation in this chapter, 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. 30. 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 12 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

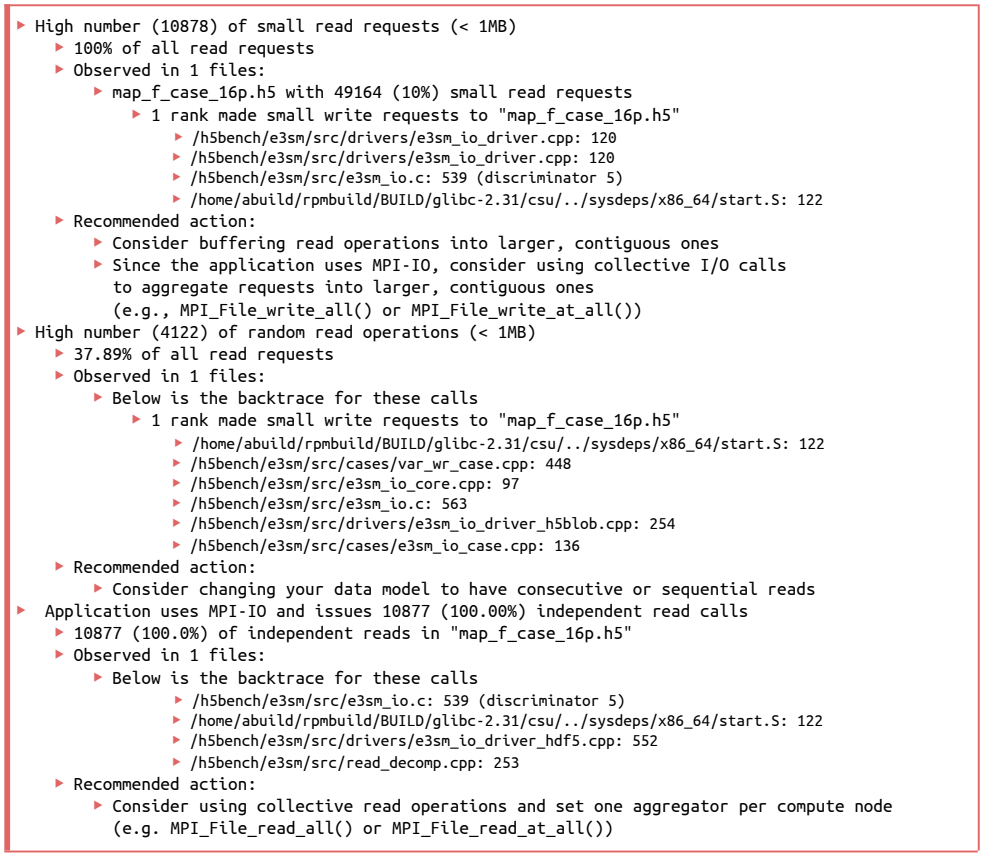


Figure 30. Critical issues detected by *Drishti* for the baseline (run-as-is) execution of E3SM in Perlmutter (NERSC) with Darshan metrics/traces.

Table 12. 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

is trying 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.

4.6 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× and 6.9× 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.

By extending interactive analysis with source-level context, this chapter advances I/O diagnosis from visual inspection to precise attribution of bottlenecks in the application code. This capability not only improves the usability of trace data but also empowers developers to take targeted corrective actions. With the combination of visualization (Chapter III) and source attribution (this chapter), the first major limitation identified in Chapter I—the disconnect between tracing and optimization—is addressed. The next chapter turns to the second limitation: the high overhead of tuning parallel I/O. It presents a lightweight end-to-end runtime optimization workflow that adapts parameter choices dynamically with minimal cost.

CHAPTER V

RUNTIME I/O OPTIMIZATION THROUGH A LIGHTWEIGHT END-TO-END WORKFLOW

This chapter contains published material with co-authorship. §5.1, §5.2, §5.3, §5.4, and §5.5 include text from a paper published in PDSW'25. The work was a collaboration between the University of Oregon, Nanyang Technological University, and Lawrence Livermore National Laboratory. Co-authors of the publication include Dr. Chen Wang, Dr. Hariharan Devarajan, Dr. Hank Childs, and Dr. Kathryn Mohror. As first author, I wrote all the sections included here. My co-authors contributed valuable suggestions that improved the text and provided guidance and insight in weekly project meetings to advance the work.

5.1 Introduction

While the previous chapters focused on bridging the gap between I/O profiling and optimization through interactive visualization, cross-layer I/O profiling, and source-level analysis, a second major challenge remains: the high overhead associated with tuning parallel I/O at scale. Existing approaches provide important insights but often impose resource- and time-intensive costs that limit their adoption in production environments. For example, autotuning frameworks [8, 9, 7] attempt to identify optimal parameters across the I/O stack, but they typically require repeated runs and exhaustive searches, sometimes taking hours to converge. Machine learning–based approaches [78, 124, 144] face similar challenges, as they depend on extensive training data from multiple simulations and often lack generality across applications and libraries. Collectively, these techniques offer benefits, but their substantial overhead costs serve as barriers to adoption.

To address this limitation, this chapter introduces *SmartIO*, a lightweight runtime optimization framework that eliminates the need for prior profiling, model training, or parameter searches. *SmartIO* combines three components into an end-to-end workflow: (1) prediction of future I/O calls through detection of recurring patterns using context-free grammars (CFGs) [27], (2) extraction of I/O access patterns and insights from the predicted calls, and (3) optimization via a rules-based mapping engine constructed through literature review and empirical evaluation. This approach dynamically tunes parameters of HDF5, ROMIO, and Lustre PFS at runtime, providing adaptive optimization during execution with minimal cost.

Our evaluations show that *SmartIO* achieves significant performance gains, including approximately a 13× improvement in read bandwidth and a 12× improvement in write bandwidth for IOR [108], and nearly a 4× increase in overall I/O bandwidth for Flash-X [37]. These benefits come with minimal runtime overhead—less than one second for IOR and about four seconds for Flash-X. Comparisons with state-of-the-art tools show that *SmartIO* delivers comparable or superior performance while dramatically reducing tuning overhead. To further ensure practicality, *SmartIO* has been integrated into the Recorder I/O tracing framework [131], enabling out-of-the-box use without requiring code modifications or recompilation.

In all, this chapter presents the following key contributions:

1. A novel end-to-end workflow for predicting and optimizing I/O performance at runtime without reliance on prior model training, profiling, or parameter searches, thereby minimizing time and resource overhead.

2. A rules-based mapping engine, derived from literature review and empirical evaluation, that translates I/O insights into actionable optimizations at runtime.

3. End-to-end runtime optimization across multiple layers of the I/O stack, including HDF5, ROMIO, and Lustre.

4. A demonstration of this workflow's effectiveness on both benchmarks and real-world applications, along with a comparative study against state-of-the-art approaches.

The remainder of the chapter is organized as follows. The design of *SmartIO* is presented in §5.2. The applicability of *SmartIO* with case studies is presented in §5.3. The limitations of *SmartIO* are presented in §5.4. The conclusion and future work are presented in §5.5.

5.2 Architecture

This section explains the end-to-end workflow of *SmartIO* and details the design of its core components.

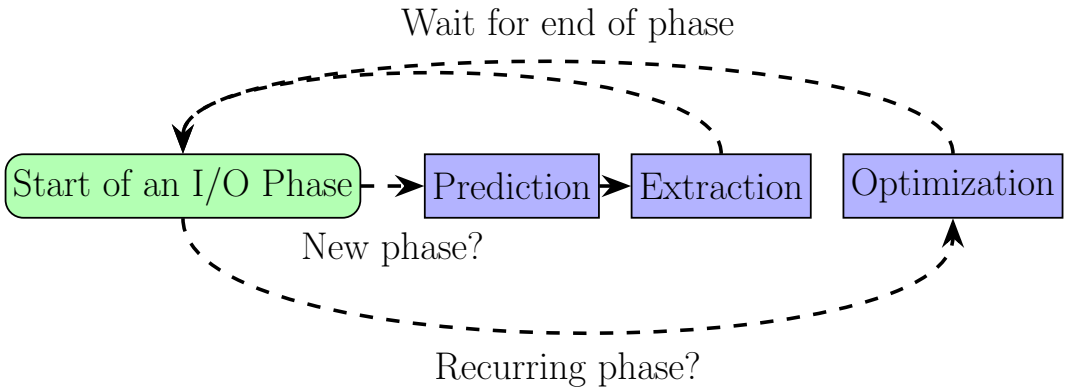


Figure 31. Overview of the *SmartIO* workflow: *SmartIO* invokes the prediction and extraction component for a new I/O phase and optimization component for a recurring I/O phase. All components operate dynamically at runtime.

5.2.1 Building the End-to-End Workflow.

As discussed briefly in §5.1, the end-to-end workflow of *SmartIO* is composed of three main components: (1) **prediction**, (2) **extraction**, and (3) **optimization**. The prediction component predicts future I/O calls through recurring pattern detection, the extraction component extracts I/O insights and access patterns from the predicted calls, and the optimization component enhances I/O performance by translating extracted insights into actionable optimizations using a rules-based mapping engine. Before jumping into the details of these components, we first discuss how these components combine at runtime to form the end-to-end workflow of *SmartIO* and, more importantly, when and how each component is invoked during the application's execution.

HPC applications often execute periodic I/O phases (such as checkpointing or data loading) throughout their runtime [25, 132, 32]. These I/O phases typically exhibit recurring patterns and account for the majority of the total I/O time. *SmartIO* builds on this observation by predicting and extracting the I/O behavior during the initial I/O phase and applying optimizations in subsequent phases. This goal is achieved by first detecting the start of an I/O phase at runtime. Once the phase is identified, *SmartIO* invokes the prediction and extraction components to learn the I/O insights of this phase. Next, when a new I/O phase starts, *SmartIO* determines whether it is a recurring or a new I/O phase. If it is a recurring I/O phase, the optimization component is invoked and optimizations are applied based on the insights collected from the last occurrence of this phase. Otherwise, the prediction and extraction components are re-invoked to learn the new patterns in this I/O phase. Fig. 31 presents an overview of *SmartIO's* workflow and the different trigger points for each component. In the following subsections, we describe the methodology used

to (1) detect the start of an I/O phase at runtime, (2) distinguish between new and recurring I/O phases, and (3) apply runtime optimizations.

5.2.1.1 Identifying the Start of an I/O Phase. Determining the starting point of an I/O phase is a challenging task, as it requires analyzing the intercepted HDF5, MPI-IO, and POSIX function calls at runtime to identify whether a specific call indicates the beginning of an I/O phase. To address this, we looked at various calls, such as `MPI_File_open`, `POSIX open`, and `H5Fcreate` to assess if they indicate the start of an I/O phase. Our analysis showed that the `MPI_File_open` and the `POSIX open` call are not accurate in determining the start of an I/O phase, as these calls were repeated multiple times in a single I/O phase. This made it difficult to access which call indicated the start of an I/O phase at runtime. However, `H5Fcreate` was called only once at the start of each I/O phase to create the HDF5 files. Therefore, we use each occurrence of `H5Fcreate` to mark the start of a new I/O phase.

5.2.1.2 Distinguishing Between New and Recurring I/O Phases. Once *SmartIO* identifies the start of an I/O phase, it determines whether the phase is recurring or new. HPC applications often exhibit interleaved I/O phases; for example, Flash-X—a highly composable software system—performs I/O through both checkpoint and plot files, each of which may exhibit distinct access patterns. Accurately distinguishing between new and recurring phases is therefore essential for effective optimization. *SmartIO* achieves this goal by keeping track of all the I/O file types (e.g., checkpoint, plot files, etc.). It does that by extracting the file name from the `H5Fcreate` call using a string-matching algorithm. Once a new phase starts, *SmartIO* determines the file to which this phase will perform I/O. If the file has been accessed before, the phase is treated as recurring; otherwise, it is classified as

new. For new phases, *SmartIO* invokes the prediction and extraction components to predict and learn I/O behavior. For recurring phases, it bypasses prediction and extraction and applies optimizations based on the insights learned from the previous occurrence of this phase.

5.2.1.3 Applying Optimizations. Once the optimization component is triggered for an I/O phase, HDF5, ROMIO, and Lustre optimizations are applied using a rules-based mapping system. HDF5 optimizations, which involve setting the alignment and metadata cache size via *fapl_id* and data transfer mode via *dxpl_id*, are applied easily, as these ids are extracted from the `H5Fcreate` and `H5open` calls. ROMIO and Lustre optimizations are applied the next time this I/O phase is encountered due to their dependency on different runtime calls. ROMIO optimizations require modifying the *MPI_Info* object, which is accessed via the `H5Pset_fapl_mpio` call, but since this call occurs before `H5Fcreate`, which was used to determine the start of an I/O phase, updates are made the next time `H5Pset_fapl_mpio` is intercepted when this I/O phase is encountered again. Similarly, Lustre optimizations, such as changing the stripe count using the *lfs setstripe* command, are also applied when this I/O phase is encountered again, as the stripe count for the current file cannot be changed once created during `H5Fcreate`.

Once it is determined that a round of optimizations needs to be performed, the HDF5, ROMIO, and Lustre optimizations are applied using the optimization rules discussed previously. However, due to their dependence on different runtime calls, the HDF5 optimizations are applied in the first round of optimization, and ROMIO and Lustre optimizations are applied in the second round of optimization for a specific file type. The HDF5 optimizations need the *fapl_id* (File access property list identifier), extracted from `H5Fcreate`, to set the alignment and metadata cache

size and the *dxpl_id* (Dataset transfer property list identifier), extracted from `H5open`, to set the data transfer mode. Since `H5Fcreate` is the trigger point for prediction or optimization, the *fapl_id* can be easily extracted from the `H5Fcreate` call to change the alignment and metadata cache values if the optimization decision is made. The `H5open` call comes after `H5Fcreate`, making it possible to apply all the HDF5 optimizations in the first round.

However, applying the ROMIO and Lustre optimizations is more complex. ROMIO optimizations are applied using `MPI_Info_set`, which requires the *MPI_Info* object to update or add new MPI hints. To get the *MPI_Info* object, the `H5Pset_fapl_mpio` call is intercepted, which stores the MPI communicator information and also includes the MPI hints being passed to the communicator. The ROMIO hints can be applied at runtime by gaining access to this *MPI_Info* object, adding or modifying existing hints, and passing those back to the communicator. However, `H5Pset_fapl_mpio` is invoked before `H5Fcreate`, meaning that when the first optimization round is triggered at `H5Fcreate`, the required *MPI_Info* object has already been set, preventing immediate modifications. Therefore, these updates are made the next time `H5Pset_fapl_mpio` is intercepted. Similarly, for the Lustre optimizations a system call is issued at `H5Fcreate` to change the stripe count using *lfs setstripe* command, however at `H5Fcreate`, the file for the current checkpoint has already been created, and hence its stripe count cannot be changed. Therefore, the stripe count for the future checkpoint files is changed at this point.

5.2.2 Prediction Component. This component uses the recurring patterns detected by Recorder's CFG to predict future I/O calls from the current point of execution.

5.2.2.1 Pattern-Recognition. Recorder leverages CFGs and the Sequitur algorithm [88] to identify recurring I/O patterns in the application code. Since *SmartIO* builds on this mechanism, we provide a high-level overview here—full details are in [133]. A CFG is a formal grammar with production rules of the form $A \rightarrow a$, where A is a single non-terminal symbol, and a is a string of terminal and/or non-terminal symbols. Recorder constructs a per-process CFG and call signature table (CST), where each terminal symbol in the CFG maps to a unique call signature stored in the CST. The call signature includes details such as function name, parameters, thread ID, and call depth. The CST is implemented as a hash table with call signatures as keys and terminal symbols as values. As I/O calls are intercepted, Recorder checks the CST: if the call is new, a terminal symbol is assigned and added to the CFG via the Sequitur algorithm, and a new entry is created for that call signature in the CST; if it's already known, the existing symbol is reused.

Fig. 32 presents a simple I/O code within nested for loops, along with its corresponding CFG and CST. The CST contains two entries: a terminal symbol and its unique call signature. There are two rules in the CFG: S, representing the outer loop, and A, representing the inner loop. The original sequence of I/O calls can be recovered by performing recursive rule applications from the starting rule S.

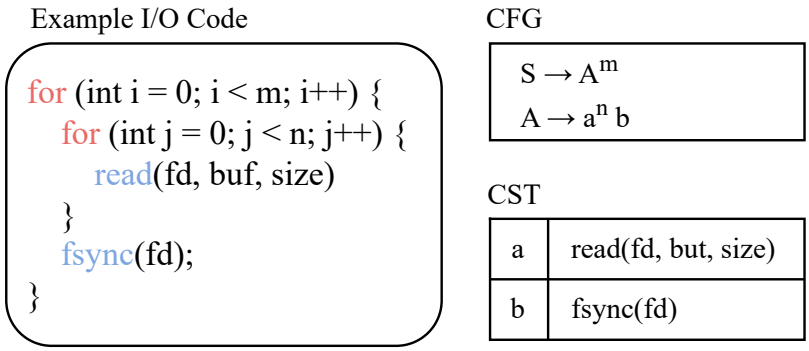


Figure 32. Example I/O code and the generated CFG and CST.

5.2.2.2 *I/O Call Sequence Prediction.*

When a new terminal symbol is generated for a call signature and is appended to the CFG, the prediction algorithm (Algorithm 2) checks if this symbol can be used to predict future I/O calls. This algorithm consists of multiple functions, including the key function `find_pattern_start`, which performs two main tasks: (1) check if the newly encountered terminal symbol is the start of a recurring pattern in the CFG (i.e., it is a starting symbol of a rule in the CFG), and (2) once the I/O call sequence is predicted, maintain a stack trace of those calls for further analysis. A function parameter determines whether `find_pattern_start` executes the first or second task.

Algorithm 2 I/O Call Sequence Prediction

```
FIND_PATTERN_START(terminal_symbol, target_rule)
if target_rule == found then
    FIND_RULE_CONFIDENCE(terminal_symbol)
    RuleConfidence * s
    HASH_FIND_INT(target_rule, s)
    max_rule_count ← s.count
    while cfg.rules do
        while rule.rule_body do
            if target_rule = sym.val then
                HASH_FIND_INT(target_rule, s)
                if s.count > max_rule_count then
                    max_rule ← s.rule
                    max_rule_count ← s.count
                end if
            else
                break
            end if
        end while
    end while
    FIND_PATTERN_START(terminal_symbol, max_rule)
    EXTRACT_IO_BEHAVIOR(stack)
end if
```

At the beginning of the prediction algorithm, the `find_pattern_start` function identifies whether the current terminal symbol can initiate a prediction. It scans all

CFG rules except the starting rule S, and if a rule begins with a terminal symbol that matches the current symbol, that rule is marked as a candidate for predicting future I/O calls and returned. Since multiple rules may share the same starting symbol, the function returns all matching rules, resulting in multiple possible predictions.

To select the most accurate prediction, the algorithm uses the `find_rule_confidence` function, which tracks the frequency of each rule within the starting rule S using a hashtable. The reason why the frequency is counted from the starting rule S is that it is the foundational rule through which all other rules originate, making it a reliable reference point for an accurate frequency count. In a scenario where there are multiple rules marked as potential candidates for prediction, the hashtable resolves this deadlock by identifying the most frequently occurring rule from the candidates, which is then chosen as the final prediction.

Once a rule has been chosen as the final prediction from the hashtable, the `find_pattern_start` function is called again to extract all the terminal and non-terminal symbols from this predicted rule. These symbols are then translated into their corresponding function calls, while a stack trace is maintained to track the call sequence. However, maintaining an accurate stack trace can be challenging due to the large and complex CFGs in HPC applications, which often include nested rules and exponential structures. To handle these complexities, a recursive mechanism is introduced within the `find_pattern_start` function. This function checks whether the predicted rule contains a non-terminal symbol within its body (another rule). If it does, the recursive function retrieves the terminal and non-terminal symbols of that nested rule. The process continues recursively, with the function retrieving terminal and non-terminal symbols from each subsequent rule, until it encounters a rule composed solely of terminal symbols.

Table 13. List of all the function calls across different I/O libraries scanned in the predicted I/O call stack trace to extract runtime I/O insights.

	Insight	Function Signature	Function Parameter	Inter-process communication
HDF5	Dataset creation property list ID	H5Pcreate	*H5P_class_t* type	✗
	File access property list ID	H5Fcreate	*hid_t* access_id	✗
	File name	H5Fcreate	*const char** name	✗
	File operation	N/A	N/A	✓
MPI-IO	Collective write	MPI_File_write_at_all	N/A	✓
	Independent write	MPI_File_write_at	N/A	✓
	Collective read	MPI_File_read_at_all	N/A	✓
	Independent read	MPI_File_read_at	N/A	✓
POSIX	Transfer size	write/read	*size_t* count	✗
	Spatial Locality	write/read	*off_t* offset	✗

5.2.3 Extraction Component. This component extracts insights from the I/O calls predicted by the first component. To extract these insights, the stack trace of the predicted I/O call sequence is scanned to identify key function signatures. For instance, to determine whether the future I/O calls will use collective or independent I/O, the component detects specific MPI-IO function calls in the stack, such as `MPI_File_write_at_all`, `MPI_File_write_at`, `MPI_File_read_at_all`, and `MPI_File_read_at`. Similarly, to get the transfer sizes of future I/O calls, the algorithm scans for specific POSIX-IO functions such as `read` and `write`. In such cases, simply identifying the function call is insufficient, as the transfer size information is the *count* parameter of the function. Therefore, the algorithm also extracts the value of this parameter to retrieve the required transfer size information.

Some I/O insights, such as file operation mode, require a unified view of values across all processes. For instance, to determine whether the future I/O calls will use a file-per-process or shared-file approach, the algorithm examines the file name extracted from the `H5Fcreate` call across all processes. If all the processes write to or read from the same file, then the file operation is set to shared-file; however if each process writes to a distinct file, the file operation is set to file-per-process. To do

this, the extraction algorithm uses MPI to implement an inter-process communication routine. This routine uses MPI collective calls such as `MPI_Gather` and `MPI_Bcast` to efficiently collect and broadcast file name information across processes and determine the correct file operation mode. Table 13 provides a detailed list of all insights that this component extracts from the predicted I/O call sequence at runtime, along with the key function signatures, their parameters, and the corresponding I/O libraries from which insights are extracted. Additionally, it also specifies whether gathering a particular insight requires any inter-process communication.

Table 14. The optimization rules for HDF5, ROMIO, and Lustre, along with their corresponding sources from the literature, and an indication of whether each rule was empirically evaluated.

Optimization	Rule	Evaluation
HDF5 Data Transfer Mode	**1.** Use independent I/O for sequential reads/writes to a shared file [28, 70]	✗
	2. Use collective I/O for random or non-sequential reads/writes to a shared file [28, 121, 122, 120, 79]	✗
	3. For file-per-process configurations, the data transfer mode should be left unchanged	✓
HDF5 Alignment	**4.** Set alignment between 1-16MB if transfer size $<$ 16MB, otherwise set it $>=$ 16MB [142]	✗
HDF5 Metadata Cache	**5.** Set metadata cache size between 1-16MB if transfer size $<$ 16MB, otherwise set it $>=$ 16MB [122]	✗
ROMIO Collective Buffering	**6.** Use *cb_config_list* to increase aggregators per node when performing collective buffering	✓
	7. Disable *romio_cb_write* for sequential accesses to a shared file, otherwise keep default [70]	✓
ROMIO Data Sieving	**8.** Keep default values for the data sieving parameters [120]	✓
Lustre Stripe Count	**9.** For file-per-process configurations, set stripe count to 1 [90, 83, 126, 4, 69]	✗
	10. For a shared file, set a progressive stripe layout if Lustre version $>$ 2.10; otherwise, set stripe count according to file size [90, 83, 126, 4, 69, 107, 103]	✓

5.2.4 Optimization Component. This component maps the extracted insights to optimizations for HDF5, ROMIO, and Lustre PFS using a rules-based mapping system. These optimization rules, derived through extensive literature review and empirical evaluation, serve as a decision framework to match I/O insights with the most effective optimizations. This approach allows *SmartIO* to select optimal values for various parameters at runtime, including HDF5 data transfer mode, alignment, and metadata cache size; ROMIO collective buffering and data sieving; and Lustre PFS stripe settings. By adapting the parameter values at runtime, *SmartIO* ensures optimal I/O performance without requiring prior model training, profiling, or parameter searches. The 10 rules governing these optimizations and their supporting evidence from literature and empirical evaluation are presented in Table 14. The next sub-sections detail the empirical evaluations conducted for rules where literature-based reasoning was unavailable.

5.2.4.1 Evaluations for HDF5 Data Transfer Mode Rules. Table 14 shows that sufficient literature was available to establish the optimization rule for the data transfer mode in a shared file configuration; however, for the file-per-process configuration, we could not find any concrete study that investigated this scenario. As a result, we conducted an empirical evaluation with the IOR benchmark, running it with 32 compute nodes, 1024 processes, 512MB block, and 64KB transfer size. Both collective and independent I/O modes were tested under this configuration. The results showed that collective and independent I/O performed equally well, with independent I/O showing only a marginal 3.17% average decrease in total execution time compared to collective I/O. Given the negligible difference in performance for these two modes in file-per-process configuration, the optimization rule for this case leaves the data transfer mode unchanged.

5.2.4.2 Evaluations for ROMIO Rules. There was insufficient literature available to formulate optimization rules for ROMIO parameters. Therefore, an empirical evaluation was conducted using IOR to derive these rules by iteratively testing various ROMIO hints on different sets of values. This testing was performed with the IOR benchmark using 128MB block size, 4MB transfer size, 8 compute nodes, and 400 processes. Table 15 presents the results of tuning some of the collective buffering hints. The results show that changing the number of aggregators per node from the default positively impacts the I/O performance, with the best performance observed with 8 aggregators per node. For the *cb_buffer_size* parameter, the execution time decreased slightly when *cb_buffer_size* was set to 2MB, however, the performance improvement was not substantial enough to recommend changing the default *cb_buffer_size* value. Changing the default value of the *romio_cb_write* parameter led to a substantial decrease in total execution time. By disabling *romio_cb_write*, the execution time decreased by approximately 76% as compared to when *romio_cb_write* was set to automatic, likely due to the elimination of the collective buffering overhead [70]. However, it is important to note that these experiments were conducted with sequential and shared-file access patterns, where we have already established that independent I/O performs better. We did not evaluate *romio_cb_write* for non-sequential data accesses, but based on prior studies [120], we expect that enabling or keeping the default value of this parameter in such cases would improve performance. Our evaluations with the data sieving parameters showed little to no effect on I/O performance, hence, the optimization rules keep the default value for those parameters.

Table 15. Effect of changing the default ROMIO collective buffering parameters on I/O performance.

ROMIO Hint	Value	% ± in execution times from default
cb_buffer_size (Default=16MB)	8MB	−0.06%
	4MB	+2.47%
	2MB	−3.89%
cb_config_list (Default=1)	*:2	−59.54%
	*:4	−54.44%
	*:8	−72.34%
	*:16	−70.46%
romio_cb_write (Default=automatic)	enable	+1.30%
	disable	−76.23%
romio_cb_read (Default=automatic)	enable	+661.52%
	disable	−0.13%

This paper [120] tested the data sieving and collective buffering performance on different applications. It also mentions that the default values of some of the data sieving parameters such as *ind_wr_buffer_size* are already set to optimal.

The ROMIO collective buffering optimizations are applied when collective data transfer is selected at the HDF5 level, as determined by the predicted insights and the corresponding HDF5 data transfer rules. These optimizations are also applied when the predicted insights and the corresponding HDF5 data transfer rules are not able to determine the data transfer mode, as we assume that the application will assume some level of collective buffering, even if our framework was not able to detect that. For the data sieving optimizations, we did not observe any significant performance gains when tuning the data sieving parameters. Consequently, the default values of these parameters are used.

5.2.4.3 Evaluations for Lustre Stripe Count Rules. The performance of Lustre PFS is heavily dependent on the stripe settings [103]. Two main parameters control striping in Lustre, determining how data is distributed across

storage resources: *stripe count* and *stripe size*. In this work, we focus on optimizing the Lustre stripe count at runtime, as choosing the right stripe count can have a significant impact on the I/O performance [103].

To define the rule that chooses the most optimal stripe count at runtime using the predicted insights, we conducted an extensive review of the research literature on this topic. There are many guidelines on this topic, such as the one provided by the University of Tennessee Knoxville [90], NERSC [83], the University of Alaska Fairbanks Research Computing Systems [126], and Amazon Web Services [4]. All these guidelines have similar instructions when it comes to setting the stripe count. They recommend using a stripe count of 1 for file-per-process configurations to limit OST contention when dealing with a large number of files/processes. For a shared file, these guidelines recommend setting the stripe count according to the file size and the number of processes, and as the file size increases, the stripe count should also be increased. Another research [107] comprehensively studies the Lustre I/O performance on Hazel Hen supercomputer [19], showing an increase in write performance for large files and processes with an increase in stripe count. This study [69] provides a practice guideline for heavy I/O workloads with Lustre PFS on the supercomputers used at the Texas Advanced Computing Center (TACC). It recommends using a single stripe count for small files and a large stripe count for large files. Similarly, another work [103] showed a 239% improvement in write performance with large stripe counts.

As shown in Table 14, significant literature is available to guide us in formulating the optimization rule for choosing the correct stripe count. However, how the stripe count is set, statically or through a progressive layout, can have a significant effect on the I/O performance. Only one study [99] has evaluated progressive and static

striping; hence, we performed a scaling study with these two approaches on the IOR benchmark with block sizes of 32MB, 64MB, 128MB, and 256MB, while keeping the transfer size, compute nodes, and the number of processes constant at 4MB, 32, and 1024 respectively. In the static striping method, the stripe count was set to maximum (-1). In contrast, the progressive layout used the following striping configuration: -E 256M -c 1 -E 4G -c 4 -E -1 -c -1. This configuration defines the layout as follows:

- The first component (-E 256M -c 1) indicates a stripe count value of 1 for files up to 256M in size.
- The second component (-E 4G -c 4) indicates a stripe count value of 4 for files up to 4G in size.
- The third component (-E -1 -c -1) indicates a stripe count value of -1 (maximum) for files greater than 4G in size.

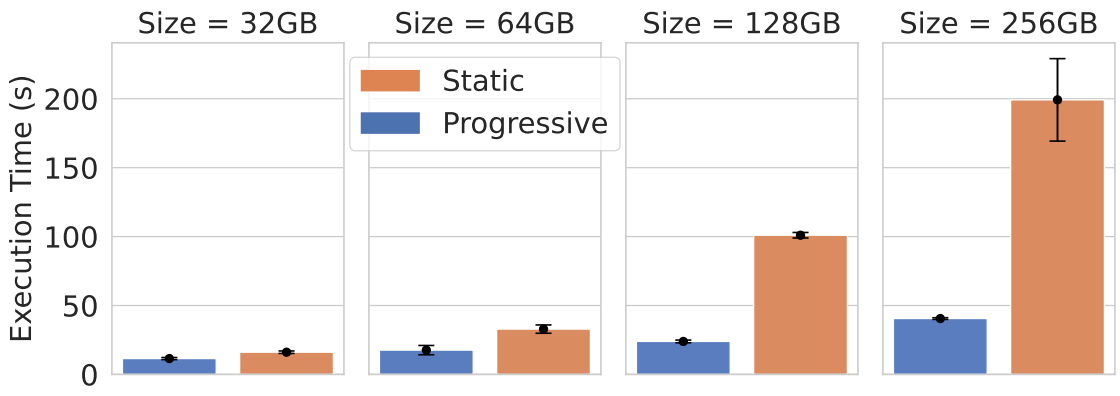


Figure 33. I/O performance comparison of progressive and static striping; execution time grows faster with static striping as the file size increases.

Fig. 33 presents the results of the scaling study, showing that progressive striping consistently outperforms static striping, with execution times increasing more rapidly under the static configuration as file size grows. The exact reason for this performance

difference, particularly for large file sizes, remains unclear. However, progressive striping's ability to adjust the stripe count and size as the file grows might be the potential reason for this difference. Furthermore, these results align with the findings presented in [99].

5.3 Evaluation

This section showcases the effectiveness of *SmartIO* using a benchmark, IOR [108], and a real-world application, Flash-X [37]. Evaluations were performed on two supercomputers at Lawrence Livermore National Laboratory: Ruby and Lassen. Ruby has 1512 Intel Xeon CLX-8276L compute nodes connected to the Lustre PFS, while Lassen has 795 IBM Power9 compute nodes connected to General Parallel File System (GPFS).

5.3.1 IOR. The IOR benchmark is widely used in the HPC I/O community to assess the performance of PFSs by simulating various access patterns. To evaluate *SmartIO*, we conducted a scaling study using the IOR benchmark with the following configuration: HDF5 API, single shared-file access, collective I/O mode, a block size of 128MB, a transfer size of 4MB, and 7 timesteps. To systematically scale the benchmark, we progressively doubled the number of compute nodes and processes, starting from 8 nodes and 256 processes, up to 32 nodes and 1024 processes. Each IOR experiment had two cases: a baseline case (without *SmartIO*) and an optimized case (with *SmartIO*). *SmartIO* predicted and learned the I/O behaviors in the first I/O phase (i.e., timestep 0 of the IOR run) and applied the optimizations in the subsequent I/O phases (i.e., timestep 1 to 6) of the execution. The insights derived from the predictions at timestep 0 were as follows:

1. **Data Transfer Mode:** Future calls will use collective I/O
2. **Transfer Size:** Each transfer size will be ¡ 4MB

3. **Shared File:** All processes will perform I/O to a shared file
4. **Access Pattern:** All processes will access sequentially
5. **File Name:** The shared file name for I/O operations
6. **File System:** The underlying file system will be Lustre on Ruby and GPFS on Lassen

Notably, the insights derived by *SmartIO* from predicted future I/O calls matched exactly with the IOR configuration presented in the previous paragraph. This demonstrates the accuracy of *SmartIO's* prediction and analysis, as neither Recorder nor *SmartIO* embedded within it had prior knowledge of the IOR configuration being used. Instead, these insights were extracted independently using the prediction and extraction components. *SmartIO* mapped the extracted insights to targeted optimizations using the rules-based mapping engine at runtime, resulting in five optimizations—three for HDF5, one for ROMIO, and one for Lustre (Ruby only)—as summarized in Table 16.

Table 16. Mapping of insights to Table 14 rules and applied optimizations before timesteps 1 and 2.

Layer	**Insight → Rule**	**Optimization**
HDF5	2 → 4 2 → 5 1, 3, 4, 5 → 1	**(1)** Changed from no **alignment** to 1MB **(2)** Changed 2MB **metadata cache size** to 8MB **(3)** Switched **data transfer mode** from collective to independent
ROMIO	4 → 7	**(4)** Switched **romio_cb_write** from automatic to disable
Lustre (Ruby)	3, 5, 6 → 10	**(5)** Set **progressive striping** with: -E 256M -c 1 -E 4G -c 4 -E -1 -c -1

Fig. 34 presents the real-time read and write bandwidth speedup relative to the baseline for each timestep (I/O phase) using *SmartIO* on Ruby and Lassen. This

figure shows that *SmartIO* significantly improved IOR read and write bandwidth as soon as the optimizations were applied before timestep 1. In some cases on Ruby, the speedup dropped a little after timestep 1, but still remained significant and consistent throughout the rest of the timesteps.

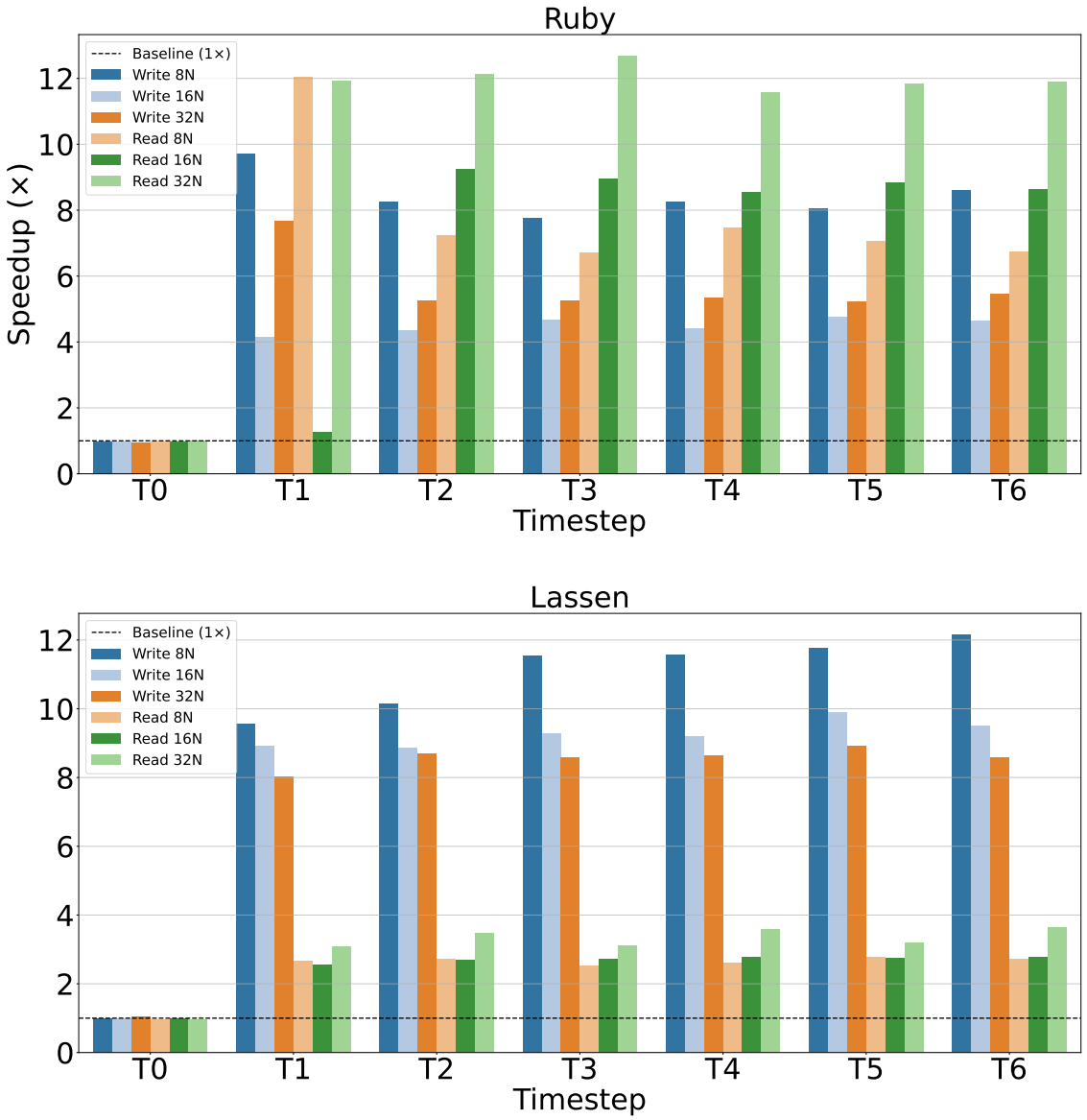


Figure 34. Real-time read and write bandwidth speedup relative to the baseline at each timestep on Ruby and Lassen using *SmartIO* on IOR with 8, 16, and 32 nodes (256, 512, and 1024 MPI processes, respectively).

Fig. 35 shows the reduction in overall execution time of the IOR benchmark achieved by *SmartIO*. The figure also highlights the execution time reduction across varying numbers of nodes and processes on each machine. On Ruby with 8 nodes and 256 processes, the execution time decreased by approximately 4×. When the compute nodes and processes were increased to 32 nodes and 1024 processes, the execution time decreased by approximately 3.3×, showing a consistent performance improvement even as the benchmark scaled. Lassen shows a similar trend.

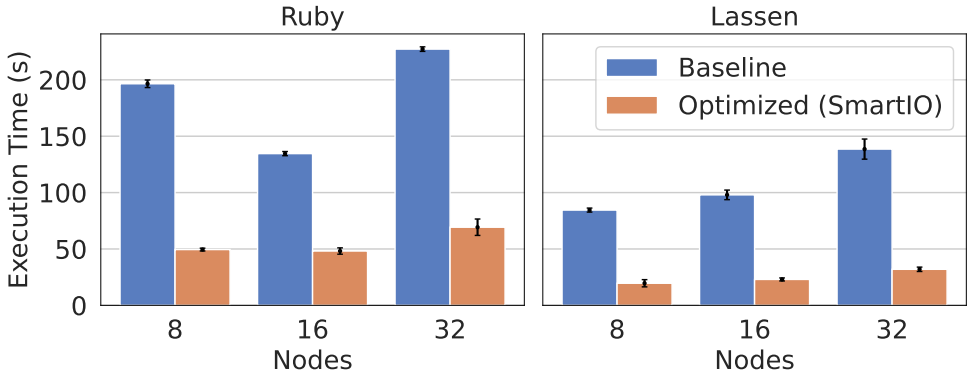


Figure 35. Comparison of baseline and optimized IOR execution time on Ruby and Lassen with 8, 16, and 32 nodes.

Out of the 5 optimizations listed in Table 16, optimizations 3, 4, and 5 accounted for the majority of the performance improvements. Notably, optimization 5 proved particularly effective at larger scales, where increased file sizes benefited significantly from a higher stripe count—a behavior previously examined in section 5.2. Optimizations 1 and 2—alignment adjustment and metadata cache size increase—did not lead to significant performance gains in our experiments. This could be attributed to the application's I/O patterns already being aligned or a relatively low volume of metadata operations.

5.3.2 Flash-X. Flash-X is an open-source, highly composable software system that can simulate various physical phenomena across various scientific domains

[37]. Here, we have demonstrated the efficacy of *SmartIO* using the Flash-X Sedov blast wave simulation, a standard simulation in the Flash-X code to verify its performance and accuracy in modeling three-dimensional hydrodynamic phenomena.

To define the runtime parameters of the Flash-X simulation, we used a parameter file to define key parameters such as grid resolution, I/O settings, and refinement levels. For our experiments, the file configured the simulation to run for 60 steps, generating 7 checkpoint files. It also sets the maximum adaptive mesh refinement (AMR) level to 6 and defined each simulation to operate in 3D with $32 \times 32 \times 32$ cells per block.

To evaluate the effectiveness of *SmartIO* across different computational scales, we ran three Flash-X Sedov simulations, each with increasing block sizes and corresponding compute resources. Specifically, we used: (1) 8 nodes with 448 processes and a block size of $8 \times 8 \times 8$, (2) 16 nodes with 896 processes and a block size of $16 \times 16 \times 16$, and (3) 32 nodes with 1792 processes and a block size of $32 \times 32 \times 32$. All experiments were performed on Ruby.

As with IOR, the scaling study with Flash-X evaluated two cases: a baseline case (without *SmartIO*) and an optimized case (with *SmartIO*). The insights derived at the first I/O phase (i.e., checkpoint 0) are shown below, and the optimizations applied to subsequent I/O phases after mapping these insights to the specific rules are shown in Table 17.

1. **Data Transfer Mode:** Future calls will utilize both collective and independent I/O
2. **Transfer Size:** Each transfer size will be $\leq$ 1MB
3. **Shared File:** All processes will perform I/O to a shared file
4. **Access Pattern:** All processes will access non-sequentially

5. **File Name:** The shared file name for I/O operations

6. **File System:** The underlying file system will be Lustre

Table 17. Mapping of insights to Table 14 rules and applied optimizations before checkpoint 1 and 2.

Layer	Insight → Rule	Optimization
HDF5	2 → 4 2 → 5	**(1)** Changed from no **alignment** to 1MB **(2)** Changed 2MB **metadata cache size** to 8MB
ROMIO	1 → 6	**(3)** Increased **cb_config_list** from 1 to 8
Lustre	3, 5, 6 → 10	**(4)** Set **progressive striping**: -E 256M -c 1 -E 4G -c 4 -E -1 -c -1

Fig. 36 presents the real-time performance speedup relative to the baseline gained by *SmartIO* on Flash-X Sedov. As can be seen from the results, *SmartIO* delivered a consistent bandwidth speedup even as the scale and complexity of the application increased. The maximum speedup of approximately $\sim 4\times$ is achieved on 8 nodes and remains consistent across all I/O phases (checkpoints). On 16 nodes, the speedup decreased to around $2\times$, but increased again to over $3\times$ on 32 nodes.

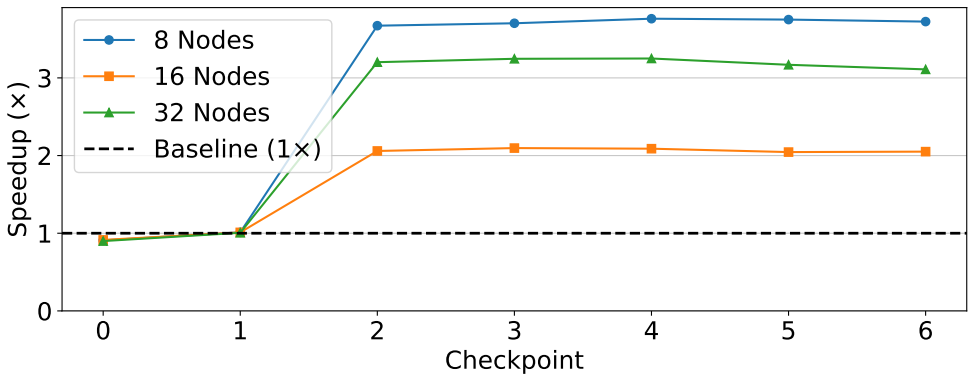


Figure 36. Real-time I/O bandwidth speedup relative to the baseline at each checkpoint on Ruby using *SmartIO* on Flash-X with 8, 16, and 32 nodes (448, 896, and 1792 MPI processes, respectively).

Fig. 37 shows the reduction in I/O time of Flash-X achieved by *SmartIO*. The I/O time was calculated by finding the time between each checkpoint open and close,

which was then summed up for all the checkpoints. *SmartIO* reduced the overall I/O time for Flash-X by over 50% and 47% on 8 and 32 nodes, respectively, demonstrating its effectiveness across increasing computational scales.

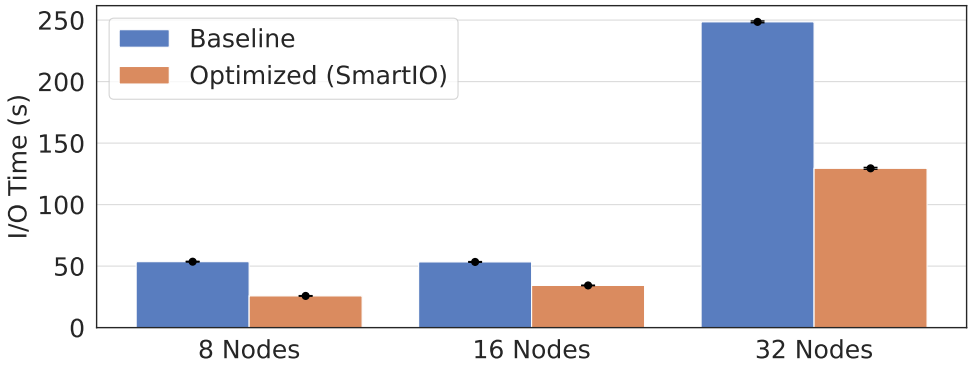


Figure 37. Comparison of baseline and optimized Flash-X I/O time on Ruby with 8, 16, and 32 nodes.

As for the optimizations listed in Table 17, optimizations 3 and 4 accounted for the majority of the performance gains discussed earlier. Notably, applying these optimizations individually did not yield significant improvements; rather, it was their combination that led to meaningful performance benefits. As was the case with IOR as well, optimizations 1 and 2—alignment adjustment and metadata cache size increase—did not yield any performance improvements.

5.3.3 Overhead. *SmartIO* incurs minimal runtime overhead. Most of this overhead originates from the prediction and extraction components, which analyze the CFG and perform associated computations. In contrast, the optimization component contributes negligibly to the overall overhead. Fig. 38 presents the overhead measured in our largest-scale experiments on both IOR and Flash-X. For IOR, the overhead was evaluated using 32 compute nodes and 1024 processes, while for Flash-X, the overhead was evaluated using 32 compute nodes, 1792 processes, and $32 \times 32 \times 32$ block size.

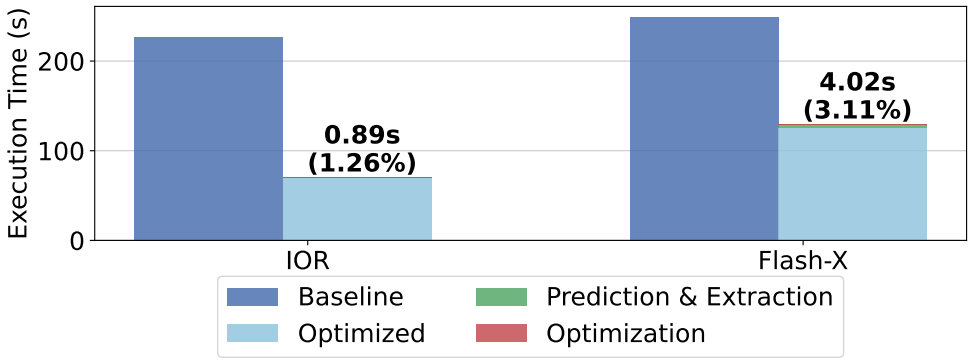


Figure 38. Overhead breakdown of the three components of *SmartIO*: prediction, extraction, and optimization

As shown in Fig. 38, the combined overhead of *SmartIO* on IOR was just 0.89s, which constituted only 1.26% of the total optimized execution time. The bulk of this overhead (0.88s) came from the prediction and extraction components, while the optimization component had a negligible overhead (0.006s). With this minimal overhead, *SmartIO* reduced the total execution time by 69.5% compared to the baseline.

For Flash-X, the total overhead slightly increased to 4.02s, representing 3.11% of the total optimized execution time. As with IOR, most of the overhead was due to the prediction and extraction components (4s), with the optimization component still contributing minimally (0.022s). With this minimal overhead, *SmartIO* reduced the overall I/O execution time by 48%. The slightly higher overhead for Flash-X was due to its more complex I/O patterns and CFGs, which required additional time for analysis.

5.3.4 Comparison with the state-of-the-art. We also compare the speedup and the tuning overhead of *SmartIO* with some state-of-the-art I/O optimization tools, such as the autotuning framework presented in [9], [8]. The

autotuning framework uses genetic algorithms and I/O performance modeling to find the most optimal tuning configurations for the different layers of the I/O stack.

Table 18. Comparison of achieved speedup and tuning overhead on the IOR benchmark: *SmartIO* vs. state-of-the-art

Tuning Tool	**Speedup (×)**	**Tuning Overhead (s)**
Autotuning Framework [8, 9]	9	36000
SmartIO	6	0.08

We begin by comparing the bandwidth speedup and tuning overhead of *SmartIO* with the autotuning framework. Due to the lack of documentation to run the autotuning framework, we rely on the experimental results reported in [9], which evaluated the autotuning framework using IOR. It configured the benchmark with 8 segments, write-only operations, and 32MB block and transfer size. Although [9] does not specify whether collective I/O was enabled in their experiment configuration, it reports collective buffering optimizations for these experiments. Therefore, we assume that collective I/O was enabled. To allow a fair comparison, we ran *SmartIO* on the same IOR configuration, PFS (Lustre), and cores (512) and reported both the write speedup and the tuning overhead. However, the experiments were performed on a different machine, which may introduce some variability in performance.

Table 18 summarizes the results. On the specified IOR configuration, *SmartIO* achieved a write bandwidth speedup of approximately 6× while incurring only 0.08 seconds of tuning overhead. In contrast, the referenced autotuning framework reported a higher speedup of 9×; however, it took over 10 hours to achieve this speedup, resulting in an overhead more than 450,000× greater than that of *SmartIO*. We were unable to compare Flash-X results with the autotuning framework, as it does not report any experiments conducted on Flash-X.

5.4 Limitations

Despite its effectiveness, *SmartIO* has an inherent limitation related to its rule-based design, as it uses a set of predefined rules to map performance insights to optimizations. Although these rules have been formed through extensive literature review and evaluations, their static nature means they remain unchanged across different systems and do not adapt based on system-specific variations. That said, this design also offers key advantages. Because *SmartIO* does not rely on performance modeling, profiling, or parameter searches, it introduces no tuning overhead—making it straightforward to deploy on large-scale HPC systems. The results in section 4.5 have shown that not only does *SmartIO* achieve significant performance speedup across multiple scientific applications, but it also cuts down entirely on the training time, offering a significant advantage over current approaches.

5.5 Conclusion

This chapter introduces *SmartIO*, a lightweight end-to-end workflow for runtime I/O optimization of HPC systems. *SmartIO* enhances the I/O performance of HPC applications at runtime by leveraging pattern detection techniques such as context-free grammars and a rule-based mapping engine to translate runtime insights into actionable optimizations. Unlike the state-of-the-art I/O optimization tools, *SmartIO* requires no prior profiling, model training, or exhaustive parameter searches—significantly reducing the tuning overhead. Experimental evaluations demonstrate that *SmartIO* delivers substantial runtime performance improvements, achieving up to $\sim 13\times$ and $\sim 12\times$ speedup in read and write bandwidth, respectively, on IOR, and a $\sim 4\times$ overall bandwidth improvement on Flash-X—all with only a few seconds of overhead. Comparative analysis with the state-of-the-art tools further

highlights *SmartIO's* effectiveness and lightweight nature, achieving comparable or, in some cases, superior performance with negligible tuning cost.

CHAPTER VI

CONCLUSION AND FUTURE WORK

6.1 Conclusion

The I/O subsystem remains a critical bottleneck in high-performance computing (HPC) systems, constraining both performance and scalability. While existing profiling and tracing tools expose detailed performance metrics, they fall short of translating these insights into actionable optimizations. Furthermore, state-of-the-art optimization approaches often incur substantial tuning overhead, limiting their practicality at large scale. This dissertation introduced two end-to-end complementary approaches to address these limitations and advance the state-of-the-art in performance analysis and optimization of parallel I/O in large-scale high-performance computing (HPC) systems. Chapter I presented the two high-level research questions that this dissertation sought to answer.

RQ 1. How can the gap between low-level I/O traces, the performance issues they reflect, and the optimizations required to resolve them be bridged?

RQ 2. How can the significant tuning overhead of state-of-the-art I/O optimization approaches be mitigated?

In Chapter II, we presented a comprehensive survey of existing research, tools, and methodologies for characterizing and evaluating parallel I/O performance in large-scale high-performance computing (HPC) systems. Serving as the foundation for the dissertation's subsequent contributions, this chapter first explains the structure and operation of the multi-layered HPC I/O stack in detail. It then reviews state-of-the-art I/O profiling, tracing, and optimization approaches, describing their mechanisms and highlighting their limitations. Together, these discussions establish the motivation and context for the end-to-end approaches introduced in later chapters.

In Chapter III, we discussed the research direction we pursued in the design and development of the interactive web-based log analysis tool called *Drishti*. This chapter addresses the **RQ 1** of the dissertation, and it's supporting sub-question *"How can interactive visualization of I/O logs aid in diagnosing and understanding application-level I/O behavior"*. The chapter presents the architecture and workflow of *Drishti*, demonstrating how it integrates tracing data from tools such as Darshan DXT [143] for visual exploration of I/O behavior. It describes how the framework extracts I/O patterns, provides interactive visualizations for diagnosing bottlenecks, and automatically identifies bottlenecks such as stragglers, small requests, and metadata overheads. Through evaluations on large-scale systems and representative scientific applications, *Drishti* is shown to enhance users' ability to interpret complex I/O traces and derive targeted optimizations.

Chapter IV answered the question *"How can cross-layer I/O profiling and static code analysis help identify the root causes of I/O bottlenecks?"*. It extended *Drishti* by incorporating cross-layer I/O profiling and source-code analysis to enable deeper diagnosis of performance bottlenecks. It demonstrated how metrics and traces from multiple tools—such as Darshan [21], DXT [143], and Recorder [131]—can be integrated into a unified, multi-layered view of the I/O stack, connecting high-level library calls to low-level file system behavior. The chapter also introduced a methodology for mapping runtime traces back to the application's source code, allowing developers to pinpoint the exact code regions responsible for inefficient I/O patterns. Through detailed evaluations, this chapter showed how cross-layer analysis and source-level mapping can reveal root causes of I/O inefficiencies that remain hidden in single-layer analyses, advancing the goal of actionable, end-to-end I/O performance diagnosis.

Chapter V answered the questions *"How can the I/O behavior be predicted and learned during the execution of the application?"* and *"How can runtime insights be translated into actionable optimizations on the fly with minimal overhead?"*. It introduced *SmartIO*, a lightweight runtime workflow for dynamic I/O optimization in HPC systems. *SmartIO* enabled online adaptation by predicting recurring I/O patterns, extracting access insights, and applying empirically derived optimization rules for HDF5, ROMIO, and Lustre. Evaluations using IOR and Flash-X demonstrated that *SmartIO* achieved notable performance gains with minimal overhead, effectively reducing the tuning cost of large-scale I/O optimization.

Collectively, these contributions demonstrated end-to-end approaches for I/O analysis and optimization that reduced overhead, improved usability, and enhanced performance across scientific and AI/ML workloads.

6.2 Future Work

While the frameworks developed in this dissertation provide significant advances in cross-layer analysis and runtime I/O optimization, some promising directions remain unexplored. These directions span natural extensions of this work, as well as broader, forward-looking opportunities that draw from the lessons learned throughout my Ph.D.

6.2.1 Leveraging Large Language Models (LLMs) and Autonomous AI Agents.

Large Language Models (LLMs) present a transformative opportunity for advancing HPC I/O optimization. Their ability to internalize large bodies of domain knowledge—including I/O traces, research publications, system logs, and best practices—creates the possibility of moving beyond static rule-based systems toward adaptive, context-aware, and self-improving optimization engines.

One compelling direction is replacing static rules in *SmartIO* and *Drishti* with an AI-driven agent that continuously learns from heterogeneous I/O traces and evolving system behavior. Such an agent could ingest Recorder-predicted I/O calls or Darshan DXT traces, correlate them with learned optimization rules, and dynamically recommend or apply runtime optimizations with increasing accuracy over time. A prototype of this concept is illustrated in Fig. 39, hinting at a future where LLM-powered intelligence enables self-adaptive, self-improving performance tuning.

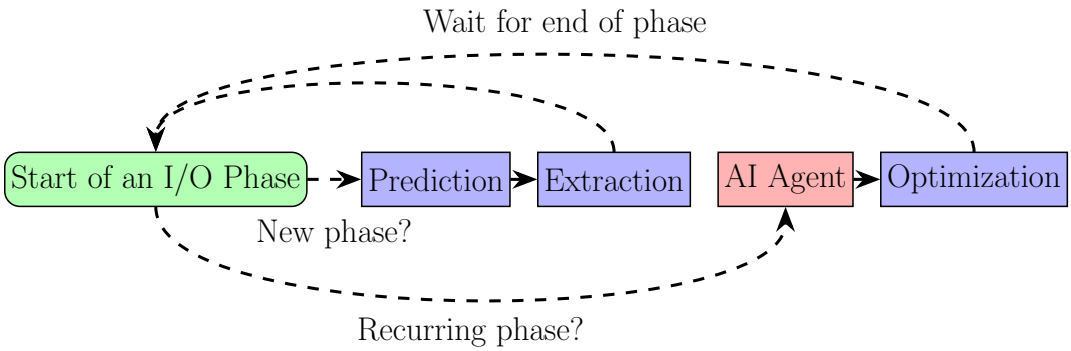


Figure 39. Future direction of incorporating AI agents.

Key research challenges include:

- **Representation learning for I/O behavior:** Designing representations that capture temporal, structural, and cross-layer dependencies in complex I/O patterns remains a fundamental and open problem.
- **Multi-modal data fusion:** HPC telemetry encompasses a wide range of heterogeneous data sources, including traces, performance profiles, and detailed file system statistics. Integrating these modalities into a unified LLM framework is non-trivial.

– **Runtime safety and stability:** Automatically applying optimizations requires extensive safeguards to prevent catastrophic mis-optimizations or regressions.

Successfully addressing these challenges would enable AI agents capable of interpreting logs, diagnosing performance anomalies, and autonomously optimizing I/O behavior during execution. This direction offers the potential to democratize performance engineering and make advanced I/O tuning accessible to scientists without deep systems expertise.

6.2.2 Expanding Library, Format, and System Coverage. This dissertation integrates metrics across multiple layers of the I/O stack, but future systems will likely rely on increasingly diverse I/O libraries, storage models, and telemetry sources. Extending the frameworks to incorporate a broader ecosystem of technologies will substantially improve applicability.

Promising directions include:

– **Additional I/O libraries:** Integrating PnetCDF, ADIOS, and UnifyFS would allow analysis of a wider class of scientific workloads.

– **Tiered and heterogeneous storage systems:** Modern HPC systems employ burst buffers, NVRAM, and multi-tier file systems. Characterizing their performance behavior remains an open research challenge.

– **Network-level telemetry:** Since many I/O bottlenecks arise from interconnect congestion, incorporating network counters and fabric-level insights can provide more comprehensive root-cause analysis.

– **File system server-side metrics:** Logs from Lustre, GPFS, and DAOS provide visibility into lock contention, RPC queue depths, and OST/MDT

imbalance. Incorporating these metrics can significantly enhance cross-layer reasoning.

Broadening the data collected by analysis frameworks allows for more accurate phase detection, better bottleneck attribution, and richer performance prediction models.

6.2.3 A Bold Future Direction: Toward Self-Optimizing and Self-Explaining HPC Systems. Looking further ahead, an ambitious yet compelling vision emerges: fully autonomous HPC systems capable of self-analyzing, self-optimizing, and self-explaining their performance behavior.

This direction extends the ideas developed in this dissertation into a holistic system-level paradigm:

- **Self-reconfiguring file systems:** Storage subsystems that automatically reorganize stripes, chunking, and object placement based on learned predictions of future access patterns.
- **Autonomous cross-application optimization:** System-wide controllers that negotiate optimization decisions across multiple applications to balance throughput, fairness, and energy.
- **Predictive I/O ecosystems:** Combining historical traces, online learning, and facility telemetry to anticipate bottlenecks before they manifest.
- **Self-explaining performance intelligence:** Systems that automatically generate human-readable explanations of performance decisions and anomalies, merging telemetry, reasoning, and policy.

While this vision is intentionally bold, its foundations lie directly in the contributions of this dissertation: cross-layer data collection, semantic interpretation

of I/O patterns, and runtime optimization of application behavior. These ideas point toward a future where HPC systems evolve from reactive infrastructures into proactive, intelligent environments.

REFERENCES CITED

[1] Ieee standard for information technology–portable operating system interface (posix(tm)) base specifications, issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pages 1–3951, 2018. doi: 10.1109/IEEESTD.2018.8277153.

[2] Megha Agarwal, Divyansh Singhvi, Preeti Malakar, and Suren Byna. Active Learning-based Automatic Tuning and Prediction of Parallel I/O Performance. In *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, pages 20–29, 2019. doi: 10.1109/PDSW49588.2019.00007.

[3] George Almási, Ralph Bellofatto, José Brunheroto, Călin Caşcaval, José G. Castaños, Luis Ceze, Paul Crumley, C. Christopher Erway, Joseph Gagliano, Derek Lieber, Xavier Martorell, José E. Moreira, Alda Sanomiya, and Karin Strauss. An overview of the blue gene/l system software organization. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, pages 543–555, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45209-6.

[4] Amazon Web Services. *Amazon FSx for Lustre Performance*, 2025. URL `https://docs.aws.amazon.com/fsx/latest/LustreGuide/performance.html`.

[5] I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, Ph. Canal, D. Casadei, O. Couet, V. Fine, L. Franco, G. Ganis, A. Gheata, D. Gonzalez Maline, M. Goto, J. Iwaszkiewicz, A. Kreshuk, D. Marcos Segura, R. Maunder, L. Moneta, A. Naumann, E. Offermann, V. Onuchin, S. Panacek, F. Rademakers, P. Russo, and M. Tadel. Root — a c++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications*, 180(12):2499–2512, 2009. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2009.08.005. URL `https://www.sciencedirect.com/science/article/pii/S0010465509002550`.

[6] Hammad Ather, Jean Luca Bez, Chen Wang, Hank Childs, Allen D. Malony, and Suren Byna. Parallel i/o characterization and optimization on large-scale hpc systems: A 360-degree survey, 2024. URL `https://arxiv.org/abs/2501.00203`.

[7] Ayşe Bağbaba. Improving collective i/o performance with machine learning supported auto-tuning. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 814–821, 2020. doi: 10.1109/IPDPSW50202.2020.00138.

[8] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Prabhat, Ruth Aydt, Quincey Koziol, and Marc Snir. Taming parallel i/o complexity with auto-tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323789. doi: 10.1145/2503210.2503278. URL https://doi.org/10.1145/2503210.2503278.

[9] Babak Behzad, Surendra Byna, Stefan M. Wild, Mr. Prabhat, and Marc Snir. Improving parallel i/o autotuning with performance modeling. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, page 253–256, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327497. doi: 10.1145/2600212.2600708. URL https://doi.org/10.1145/2600212.2600708.

[10] Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. Optimizing I/O Performance of HPC Applications with Autotuning. *ACM Trans. Parallel Comput.*, 5(4), March 2019. ISSN 2329-4949. doi: 10.1145/3309205.

[11] Jean Luca Bez, Francieli Zanon Boito, Lucas M. Schnorr, Philippe O. A. Navaux, and Jean-François Méhaut. TWINS: Server Access Coordination in the I/O Forwarding Layer. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 116–123, 2017. doi: 10.1109/PDP.2017.61.

[12] Jean Luca Bez, Francieli Zanon Boito, Ramon Nou, Alberto Miranda, Toni Cortes, and Philippe O.A. Navaux. Adaptive Request Scheduling for the I/O Forwarding Layer Using Reinforcement Learning. *Future Generation Computer Systems*, 112:1156–1169, 2020. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2020.05.005.

[13] Jean Luca Bez, Houjun Tang, Bing Xie, David Williams-Young, Rob Latham, Rob Ross, Sarp Oral, and Suren Byna. I/o bottleneck detection and tuning: Connecting the dots using interactive log analysis. In *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*, pages 15–22, 2021. doi: 10.1109/PDSW54622.2021.00008.

[14] Jean Luca Bez, Ahmad Maroof Karimi, Arnab K. Paul, Bing Xie, Suren Byna, Philip Carns, Sarp Oral, Feiyi Wang, and Jesse 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*, HPDC '22, page 43–55, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391993. doi: 10.1145/3502181.3531461.

[15] Jean Luca Bez, Suren Byna, and Shadi Ibrahim. I/o access patterns in hpc applications: A 360-degree survey. *ACM Comput. Surv.*, 56(2), sep 2023. ISSN 0360-0300. doi: 10.1145/3611007. URL https://doi.org/10.1145/3611007.

[16] Jean Luca Bez, Houjun Tang, Scot Breitenfeld, Huihuo Zheng, Wei-Keng Liao, Kaiyuan Hou, Zanhua Huang, and Suren Byna. h5bench: A unified benchmark suite for evaluating hdf5 i/o performance on pre-exascale platforms. *Concurrency and Computation. Practice and Experience*, 36(16), 4 2024. doi: 10.1002/cpe.8046.

[17] Francieli Zanon Boito, Rodrigo Virote Kassick, Philippe O.A. Navaux, and Yves Denneulin. AGIOS: Application-Guided I/O Scheduling for Parallel File Systems. In *Int. Conference on Parallel and Distributed Systems*, pages 43–50, 2013. doi: 10.1109/ICPADS.2013.19.

[18] S. Byna, M. Breitenfeld, B. Dong, Q. Koziol, Elena Pourmal, D. Robinson, Jérome Soumagne, Houjun Tang, Venkatram Vishwanath, and R. Warren. ExaHDF5: Delivering Efficient Parallel I/O on Exascale Computing Systems. *JCST*, 35:145–160, 2020. doi: 10.1007/s11390-020-9822-9.

[19] Thomas Bönisch, Michael Resch, Thomas Schwitalla, Matthias Meinke, Volker Wulfmeyer, and Kirsten Warrach-Sagi. Hazel hen – leading hpc technology and its impact on science in germany and europe. *Parallel Computing*, 64:3–11, 2017. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2017.02.002. URL https://www.sciencedirect.com/science/article/pii/S0167819117300194. High-End Computing for Next-Generation Scientific Discovery.

[20] André Ramos Carneiro, Jean Luca Bez, Carla Osthoff, Lucas Mello Schnorr, and Philippe O.A. Navaux. Uncovering I/O Demands on HPC Platforms: Peeking Under the Hood of Santos Dumont. *Journal of Parallel and Distributed Computing*, 182:104744, 2023. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2023.104744.

[21] 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)*, pages 1–10, Los Alamitos, CA, USA, sep 2009. IEEE Computer Society. doi: 10.1109/CLUSTR.2009.5289150. URL https://doi.ieeecomputersociety.org/10.1109/CLUSTR.2009.5289150.

[22] Philip Carns, Julian Kunkel, Kathryn Mohror, and Martin Schulz. Understanding I/O Behavior in Scientific and Data-Intensive Computing (Dagstuhl Seminar 21332). *Dagstuhl Reports*, 11(7):16–75, 2021. ISSN 2192-5283. doi: 10.4230/DagRep.11.7.16.

[23] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *4th Annual Linux Showcase & Conference (ALS 2000)*, Atlanta, GA, October 2000. USENIX Association. URL https://www.usenix.org/conference/als-2000/pvfs-parallel-file-system-linux-clusters.

[24] Christopher D. Carothers, Jeremy S. Meredith, Mark P. Blanco, Jeffrey S. Vetter, Misbah Mubarak, Justin LaPre, and Shirley Moore. Durango: Scalable synthetic workload generation for extreme-scale application performance modeling and simulation. SIGSIM-PADS '17, page 97–108, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344890. doi: 10.1145/3064911.3064923. URL https://doi.org/10.1145/3064911.3064923.

[25] Jesus Carretero, Emmanuel Jeannot, Guillaume Pallez, David E. Singh, and Nicolas Vidal. Mapping and scheduling hpc applications for optimizing i/o. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379830. doi: 10.1145/3392717.3392764. URL https://doi.org/10.1145/3392717.3392764.

[26] Nadav Chachmon, Daniel Richins, Robert Cohn, Magnus Christensson, Wenzhi Cui, and Vijay Janapa Reddi. Simulation and analysis engine for scale-out workloads. ICS '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343619. doi: 10.1145/2925426.2926293. URL https://doi.org/10.1145/2925426.2926293.

[27] William Y.C. Chen. Context-free grammars, differential operators and formal power series. *Theoretical Computer Science*, 117(1):113–129, 1993. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(93)90307-F. URL https://www.sciencedirect.com/science/article/pii/030439759390307F.

[28] Christian M. Chilan and Kent Yang. Performance comparison of collective i/o and independent i/o with derived datatypes. Technical report, The HDF Group, June 2006. URL https://support.hdfgroup.org/releases/hdf5/documentation/rfc/coll_ind_dd6.pdf.

[29] DWARF Debugging Information Format Committee. DWARF Debugging Information Format Version 5. URL https://dwarfstd.org/doc/DWARF5.pdf.

[30] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snirt, Bernard Traversat, and Parkson Wong. *Overview of the MPI-IO Parallel I/O Interface*, pages 127–146. Springer US, Boston, MA, 1996. ISBN 978-1-4613-1401-1. doi: 10.1007/978-1-4613-1401-1_5. URL https://doi.org/10.1007/978-1-4613-1401-1_5.

[31] Darshan Team. pyDarshan. URL https://github.com/darshan-hpc/darshan/tree/main/darshan-util/pydarshan.

[32] Hariharan Devarajan and Kathryn Mohror. Extracting and characterizing i/o behavior of hpc workloads. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 243–255, 2022. doi: 10.1109/CLUSTER51413.2022.00037.

[33] Hariharan Devarajan, Huihuo Zheng, Anthony Kougkas, Xian-He Sun, and Venkatram Vishwanath. Dlio: A data-centric benchmark for scientific deep learning applications. pages 81–91, 05 2021. doi: 10.1109/CCGrid51090.2021.00018.

[34] James Dickson, Steven Wright, Satheesh Maheswaran, Andy Herdman, Mark C. Miller, and Stephen Jarvis. Replicating hpc i/o workloads with proxy applications. In *2016 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 13–18, 2016. doi: 10.1109/PDSW-DISCS.2016.007.

[35] James Dickson, Steven A. Wright, Satheesh Maheswaran, J. A. Herdman, D. Harris, Mark C. Miller, and Stephen A. Jarvis. Enabling portable i/o analysis of commercially sensitive hpc applications through workload replication. 2017. URL https://api.semanticscholar.org/CorpusID:37609466.

[36] Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, and Rob Ross. Omnisc'io: A grammar-based approach to spatial and temporal i/o patterns prediction. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 623–634, 2014. doi: 10.1109/SC.2014.56.

[37] Anshu Dubey, Klaus Weide, Jared O'Neal, Akash Dhruv, Sean Couch, J. Austin Harris, Tom Klosterman, Rajeev Jain, Johann Rudi, Bronson Messer, Michael Pajkos, Jared Carlson, Ran Chu, Mohamed Wahib, Saurabh Chawdhary, Paul M. Ricker, Dongwook Lee, Katie Antypas, Katherine M. Riley, Christopher Daley, Murali Ganapathy, Francis X. Timmes, Dean M. Townsley, Marcos Vanella, John Bachan, Paul M. Rich, Shravan Kumar, Eirik Endeve, W. Raphael Hix, Anthony Mezzacappa, and Thomas Papatheodore. Flash-x: A multiphysics simulation software instrument. *SoftwareX*, 19:101168, 2022. ISSN 2352-7110. doi: https://doi.org/10.1016/j.softx.2022.101168. URL https://www.sciencedirect.com/science/article/pii/S2352711022001030.

[38] E3SM. Energy Exascale Earth System Model v2.1.0, Jan 2023. URL https://doi.org/10.11578/E3SM/dc.20230110.5.

[39] Luca Fedeli, Axel Huebl, France Boillod-Cerneux, Thomas Clark, Kevin Gott, Conrad Hillairet, Stephan Jaure, Adrien Leblanc, Rémi Lehe, Andrew Myers, Christelle Piechurski, Mitsuhisa Sato, Neïl Zaim, Weiqun Zhang, Jean-Luc Vay, and Henri 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*, pages 1–12, 2022. doi: 10.1109/SC41404.2022.00008.

[40] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, page 36–47, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306140. doi: 10.1145/1966895.1966900. URL `https://doi.org/10.1145/1966895.1966900`.

[41] Wolfgang Frings, Morris Riedel, Achim Streit, Daniel Mallmann, Sven Van den Berghe, David Snelling, and Vivian Li. Llview: User-level monitoring in computational grids and e-science infrastructures. 01 2007.

[42] Jim Garlick. LMT3 - Lustre Monitoring Tool, 2010. URL `https://github.com/LLNL/lmt`.

[43] Anjus George, Rick Mohr, James Simmons, and Sarp Oral. Understanding lustre internals. second edition. 9 2021. doi: 10.2172/1824954. URL `https://www.osti.gov/biblio/1824954`.

[44] Anjus George, Rick Mohr, James Simmons, and Sarp Oral. Understanding Lustre Internals 2nd Edition. 9 2021. doi: 10.2172/1824954. URL `https://www.osti.gov/biblio/1824954`.

[45] William F Godoy, Jenna Delozier, and Gregory R Watson. Modeling pre-exascale amr parallel i/o workloads via proxy applications. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, page 952–961. IEEE, May 2022. doi: 10.1109/ipdpsw55747.2022.00153. URL `http://dx.doi.org/10.1109/IPDPSW55747.2022.00153`.

[46] Jean-Christophe Golaz, Luke P. Van Roekel, Xue Zheng, Andrew F. Roberts, Jonathan D. Wolfe, Wuyin Lin, Andrew M. Bradley, Qi Tang, Mathew E. Maltrud, Ryan M. Forsyth, Chengzhu Zhang, Tian Zhou, Kai Zhang, Charles S. Zender, Mingxuan Wu, Hailong Wang, Adrian K. Turner, Balwinder Singh, Jadwiga H. Richter, Yi Qin, Mark R. Petersen, Azamat Mametjanov, Po-Lun Ma, Vincent E. Larson, Jayesh Krishna, Noel D. Keen, Nicole Jeffery, Elizabeth C. Hunke, Walter M. Hannah, Oksana Guba, Brian M. Griffin, Yan Feng, Darren Engwirda, Alan V. Di Vittorio, Cheng Dang, LeAnn M. Conlon, Chih-Chieh-Jack Chen, Michael A. Brunke, Gautam Bisht, James J. Benedict, Xylar S. Asay-Davis, Yuying Zhang, Meng Zhang, Xubin Zeng, Shaocheng Xie, Phillip J. Wolfram, Tom Vo, Milena Veneziani, Teklu K. Tesfa, Sarat Sreepathi, Andrew G. Salinger, J. E. Jack Reeves Eyre, Michael J. Prather, Salil Mahajan, Qing Li, Philip W. Jones, Robert L. Jacob, Gunther W. Huebler, Xianglei Huang, Benjamin R. Hillman, Bryce E. Harrop, James G. Foucar, Yilin Fang, Darin S. Comeau, Peter M. Caldwell, Tony Bartoletti, Karthik Balaguru, Mark A. Taylor, Renata B. McCoy, L. Ruby Leung, and David C. Bader. The DOE E3SM Model Version 2: Overview of the Physical Model and Initial Model Evaluation. *Journal of Advances in Modeling Earth Systems*, 14 (12):e2022MS003156, 2022. doi: https://doi.org/10.1029/2022MS003156.

[47] Hanisch, R. J., Farris, A., Greisen, E. W., Pence, W. D., Schlesinger, B. M., Teuben, P. J., Thompson, R. W., and Warnock, A. Definition of the flexible image transport system (fits)*. *A&A*, 376(1):359–380, 2001. doi: 10.1051/0004-6361:20010923. URL `https://doi.org/10.1051/0004-6361:20010923`.

[48] Meng Hao, Weizhe Zhang, You Zhang, Marc Snir, and Laurence T. Yang. Automatic generation of benchmarks for i/o-intensive parallel applications. *Journal of Parallel and Distributed Computing*, 124:1–13, 2019. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2018.10.004. URL `https://www.sciencedirect.com/science/article/pii/S074373151830738X`.

[49] Peter Harrington, Wucherl Yoo, Alexander Sim, and Kesheng Wu. Diagnosing parallel i/o bottlenecks in hpc applications. In *International Conference for High Performance Computing Networking Storage and Analysis (SCI7) ACM Student Research Competition (SRC)*, page 4, 2017.

[50] Edward Hartnett and Jim Edwards. The Parallel I/O (PIO) C/FORTRAN Libraries for Scalable HPC Performance. In *101st American Meteorological Society Annual Meeting (AMS)*, 01 2021.

[51] 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. openPMD: A meta data standard for particle and mesh based data, 2015.

[52] Sun Microsystems Inc. High-Performance Storage Architecture and Scalable Cluster File System. Technical report, 2007.

[53] Florin Isaila, Prasanna Balaprakash, Stefan M. Wild, Dries Kimpe, Rob Latham, Rob Ross, and Paul Hovland. Collective i/o tuning using analytical and machine learning models. In *2015 IEEE International Conference on Cluster Computing*, pages 128–137, 2015. doi: 10.1109/CLUSTER.2015.29.

[54] Mihailo Isakov, Eliakin del Rosario, Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert B. Ross, and Michel 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*, SC '20. IEEE Press, 2020. ISBN 9781728199986.

[55] Ahmad Maroof Karimi, Arnab K. Paul, and Feiyi Wang. I/O Performance Analysis of Machine Learning Workloads on Leadership Scale Supercomputer. *Performance Evaluation*, 157-158:102318, 2022. ISSN 0166-5316. doi: https://doi.org/10.1016/j.peva.2022.102318. URL `https://www.sciencedirect.com/science/article/pii/S0166531622000268`.

[56] Seong Jo Kim, Seung Woo Son, Wei-keng Liao, Mahmut Kandemir, Rajeev Thakur, and Alok Choudhary. Iopin: Runtime profiling of parallel i/o in hpc systems. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 18–23, 2012. doi: 10.1109/SC.Companion.2012.14.

[57] Seong Jo Kim, Yuanrui Zhang, Seung Woo Son, Mahmut Kandemir, Wei-Keng Liao, Rajeev Thakur, and Alok Choudhary. Iopro: A parallel i/o profiling and visualization framework for high-performance storage systems. *J. Supercomput.*, 71(3):840–870, mar 2015. ISSN 0920-8542. doi: 10.1007/s11227-014-1329-0. URL `https://doi.org/10.1007/s11227-014-1329-0`.

[58] Sunggon Kim, Alex Sim, Kesheng Wu, Suren Byna, and Yongseok Son. Design and implementation of i/o performance prediction scheme on hpc systems through large-scale log analysis. *Journal of Big Data*, 10, 05 2023. doi: 10.1186/s40537-023-00741-4.

[59] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-p: A joint performance measurement run-time infrastructure for periscope,scalasca, tau, and vampir. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31476-6.

[60] Fabian Koller et al. openPMD-api: C++ & Python API for Scientific I/O with openPMD, 2019. URL doi.org/10.14278/rodare.209.

[61] Julian Kunkel, Michaela Zimmer, and Eugen Betke. Predicting performance of non-contiguous i/o with machine learning. In Julian M. Kunkel and Thomas Ludwig, editors, *High Performance Computing*, pages 257–273, Cham, 2015. Springer International Publishing. ISBN 978-3-319-20119-1.

[62] Julian Martin Kunkel, Eugen Betke, Matt Bryson, Philip Carns, Rosemary Francis, Wolfgang Frings, Roland Laifer, and Sandra Mendez. Tools for analyzing parallel i/o. In Rio Yokota, Michèle Weiland, John Shalf, and Sadaf Alam, editors, *High Performance Computing*, pages 49–70, Cham, 2018. Springer International Publishing. ISBN 978-3-030-02465-9.

[63] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/O Performance Challenges at Leadership Scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, 2009. ACM. ISBN 9781605587448. doi: 10.1145/1654059.1654100.

[64] Rob Latham, Chris Daley, Wei-keng Liao, Kui Gao, Rob Ross, Anshu Dubey, and Alok Choudhary. A case study for scientific I/O: improving the FLASH astrophysics code. *Computational Science and Discovery*, 5(1):015001, January 2012. doi: 10.1088/1749-4699/5/1/015001.

[65] Choonghwan Lee, Muqun Yang, and Ruth A. Aydt. Netcdf-4 performance report. 2008. URL https://api.semanticscholar.org/CorpusID:15588867.

[66] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, page 39, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136951. doi: 10.1145/1048935.1050189. URL https://doi.org/10.1145/1048935.1050189.

[67] Yan Li, Oceane Bel, Kenneth Chang, Ethan L. Miller, and Darrell D. E. Long. CAPES: Unsupervised Storage Performance Tuning Using Neural Network-Based Deep Reinforcement Learning. In *SC'17*, November 2017. doi: 10.1145/3126908.3126951.

[68] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, Manish Parashar, Nagiza Samatova, Karsten Schwan, Arie Shoshani, Matthew Wolf, Kesheng Wu, and Weikuan Yu. Hello adios: the challenges and lessons of developing leadership class i/o frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014. doi: https://doi.org/10.1002/cpe.3125. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3125`.

[69] Si Liu, Lei Huang, Hang Liu, Amit Ruhela, Virginia Trueheart, Susan Lindsey, and Quan Yuan. Practice guideline for heavy i/o workloads with lustre file systems on tacc supercomputers. In *Practice and Experience in Advanced Research Computing 2021: Evolution Across All Dimensions*, PEARC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382922. doi: 10.1145/3437359.3465570. URL `https://doi.org/10.1145/3437359.3465570`.

[70] Wei Liu, Kai Wu, Jialin Liu, Feng Chen, and Dong Li. Performance evaluation and modeling of hpc i/o on non-volatile memory. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–10, 2017. doi: 10.1109/NAS.2017.8026869.

[71] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S Vazhkudai. Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems. In *SC16*, pages 819–829. IEEE, 2016. doi: 10.1109/SC.2016.69.

[72] Glenn K. Lockwood, Wucherl Yoo, Suren Byna, Nicholas J. Wright, Shane Snyder, Kevin Harms, Zachary Nault, and Philip Carns. Umami: a recipe for generating meaningful metrics through holistic i/o performance analysis. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, PDSW-DISCS '17, page 55–60, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351348. doi: 10.1145/3149393.3149395. URL `https://doi.org/10.1145/3149393.3149395`.

[73] Glenn K. Lockwood, Shane Snyder, Teng Wang, Suren Byna, Philip Carns, and Nicholas J. Wright. A year in the life of a parallel file system. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 931–943, 2018. doi: 10.1109/SC.2018.00077.

[74] Glenn K. Lockwood, Nicholas J. Wright, Shane Snyder, Philip Carns, George Brown, and Kevin Harms. Tokio on clusterstor: Connecting standard tools to enable holistic i/o performance analysis. 1 2018. URL `https://www.osti.gov/biblio/1632125`.

[75] Jay Lofstead, Milo Polte, Garth Gibson, Scott Klasky, Karsten Schwan, Ron Oldfield, Matthew Wolf, and Qing Liu. Six Degrees of Scientific Data: Reading Patterns for Extreme Scale Science IO. In *HPDC'11*, page 49–60, New York, NY, USA, 2011. ACM. ISBN 9781450305525. doi: 10.1145/1996130.1996139.

[76] Xiaoqing Luo, Frank Mueller, Philip Carns, John Jenkins, Robert Latham, Robert Ross, and Shane Snyder. Hpc i/o trace extrapolation. In *Proceedings of the 4th Workshop on Extreme Scale Programming Tools*, ESPT '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450339971. doi: 10.1145/2832106.2832108. URL https://doi.org/10.1145/2832106.2832108.

[77] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. A multiplatform study of i/o behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 33–44, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335508. doi: 10.1145/2749246.2749269. URL https://doi.org/10.1145/2749246.2749269.

[78] Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert Latham, Robert Ross, Shane Snyder, and Stefan M. Wild. Machine learning based parallel i/o predictive modeling: A case study on lustre file systems. In Rio Yokota, Michèle Weiland, David Keyes, and Carsten Trinitis, editors, *High Performance Computing*, pages 184–204, Cham, 2018. Springer International Publishing. ISBN 978-3-319-92040-5.

[79] Sandra Mendez, Sebastian Lührs, Dominic Sloan-Murphy, Andrew Turner, and Volker Weinberg. Best practice guide - parallel i/o, 2019. URL https://prace-ri.eu/wp-content/uploads/Best-Practice-Guide_Parallel-IO.pdf.

[80] OE Bronson Messer, Ed D'Azevedo, Judy Hill, Wayne Joubert, Mark Berrill, and Christopher Zimmer. Miniapps derived from production hpc applications using multiple programing models. *Int. J. High Perform. Comput. Appl.*, 32(4): 582–593, jul 2018. ISSN 1094-3420. doi: 10.1177/1094342016668241. URL https://doi.org/10.1177/1094342016668241.

[81] Nolan Monnier, Jay Lofstead, Margaret Lawson, and Matthew Curry. Profiling platform storage using io500 and mistral. In *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, pages 60–73, 2019. doi: 10.1109/PDSW49588.2019.00011.

[82] Olga Mordvinova, Dennis Runz, Julian M. Kunkel, and Thomas Ludwig. I/o performance evaluation with parabench — programmable i/o benchmark. *Procedia Computer Science*, 1(1):2125–2134, 2010. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2010.04.238. URL `https://www.sciencedirect.com/science/article/pii/S1877050910002395`. ICCS 2010.

[83] National Energy Research Scientific Computing Center. *Lustre File Striping*. Lawrence Berkeley National Laboratory, 2025. URL `https://docs.nersc.gov/performance/io/lustre/`.

[84] Sergey Nedev and Vladimir Kamenov. Hdd performance research. 01 2018.

[85] Daniel Nemirovsky, Tugberk Arkose, Nikola Markovic, Mario Nemirovsky, Osman Unsal, and Adrian Cristal. A machine learning approach for performance prediction and scheduling on heterogeneous cpus. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 121–128, 2017. doi: 10.1109/SBAC-PAD.2017.23.

[86] Sarah Neuwirth and Arnab K. Paul. Parallel i/o evaluation techniques and emerging hpc workloads: A perspective. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 671–679, 2021. doi: 10.1109/Cluster48925.2021.00100.

[87] Craig Nevill-Manning and Ian Witten. Identifying hierarchical strcture in sequences: A linear-time algorithm. *J. Artif. Intell. Res. (JAIR)*, 7:67–82, 09 1997. doi: 10.1613/jair.374.

[88] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: a linear-time algorithm. *J. Artif. Int. Res.*, 7(1):67–82, September 1997. ISSN 1076-9757.

[89] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello. VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 911–920, Los Alamitos, CA, USA, may 2019. IEEE Computer Society. doi: 10.1109/IPDPS.2019.00099.

[90] Office of Innovative Technologies. *Lustre Striping Guide*. University of Tennessee, 2025. URL `https://oit.utk.edu/hpsc/lustre-striping-guide/`.

[91] David A. Patterson. *Computer Organization and Design*. Elsevier Science; Technology, 2013.

[92] Arnab K. Paul, Olaf Faaland, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Ali 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)*, pages 202–211, 2020. doi: 10.1109/HiPC50609.2020.00034.

[93] Ivy Peng, Ian Karlin, Maya Gokhale, Kathleen Shoga, Matthew Legendre, and Todd Gamblin. A Holistic View of Memory Utilization on HPC Systems: Current and Future Trends. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '21, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450385701. doi: 10.1145/3488423.3519336. URL `https://doi.org/10.1145/3488423.3519336`.

[94] Felipe Pezoa et al. Foundations of JSON Schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273, 2016.

[95] Peter Piela. collectl Lustre Data Collection Plugins, 2021. URL `https://github.com/pcpiela/collectl-lustre`.

[96] Nicole Hemsoth Prickett. What's so bad about posix i/o? - the next platform, Apr 2018. URL `https://www.nextplatform.com/2017/09/11/whats-bad-posix-io/`.

[97] Rolf Rabenseifner, Alice Koniges, Jean-pierre Prost, and Richard Hedges. The parallel effective i/o bandwidth benchmark: b-eff-io. 12 2001.

[98] Daniel Reed, Dennis Gannon, and Jack Dongarra. HPC Forecast: Cloudy and Uncertain. *Commun. ACM*, 66(2):82–90, jan 2023. ISSN 0001-0782. doi: 10.1145/3552309. URL `https://doi.org/10.1145/3552309`.

[99] Joel Reed, Jeremy Archuleta, Michael Brim, and Josh Lothian. Evaluating dynamic file striping for lustre. 03 2015.

[100] Russ Rew and Glenn Davis. Netcdf: an interface for scientific data access. *IEEE Computer Graphics and Applications*, 10:76–82, 1990. URL `https://api.semanticscholar.org/CorpusID:11171299`.

[101] Abdulqawi Saif, Lucas Nussbaum, and Ye-Qiong Song. Ioscope: A flexible i/o tracer for workloads' i/o pattern characterization. In Rio Yokota, Michèle Weiland, John Shalf, and Sadaf Alam, editors, *High Performance Computing*, pages 103–116, Cham, 2018. Springer International Publishing. ISBN 978-3-030-02465-9.

[102] Subhash Saini, Dale Talcott, Rajeev Thakur, Panagiotis Adamidis, Rolf Rabenseifner, and Bob Ciotti. Parallel i/o performance characterization of columbia and nec sx-8 superclusters. *Parallel and Distributed Processing Symposium, International*, 0:99, 03 2007. doi: 10.1109/IPDPS.2007.370289.

[103] Subhash Saini, Jason Rappleye, Johnny Chang, David Barker, Piyush Mehrotra, and Rupak Biswas. I/o performance characterization of lustre and nasa applications on pleiades. In *2012 19th International Conference on High Performance Computing*, pages 1–10, 2012. doi: 10.1109/HiPC.2012.6507507.

[104] Karthik Sangaiah, Michael Lui, Radhika Jagtap, Stephan Diestelhorst, Siddharth Nilakantan, Ankit More, Baris Taskin, and Mark Hempstead. Synchrotrace: Synchronization-aware architecture-agnostic traces for lightweight multicore simulation of cmp and hpc workloads. *ACM Trans. Archit. Code Optim.*, 15(1), mar 2018. ISSN 1544-3566. doi: 10.1145/3158642. URL `https://doi.org/10.1145/3158642`.

[105] Jan Fabian Schmid and Julian M. Kunkel. Predicting i/o performance in hpc using artificial neural networks. *Supercomputing Frontiers and Innovations*, 3 (3):19–33, Sep. 2016. doi: 10.14529/jsfi160303. URL `https://superfri.org/index.php/superfri/article/view/105`.

[106] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002. URL `https://www.usenix.org/conference/fast-02/gpfs-shared-disk-file-system-large-computing-clusters`.

[107] Marco Seiz, Philipp Offenhäuser, Stefan Andersson, Johannes Hötzer, Henrik Hierl, Britta Nestler, and Michael Resch. Lustre i/o performance investigations on hazel hen: experiments and heuristics. *J. Supercomput.*, 77(11):12508–12536, November 2021. ISSN 0920-8542. doi: 10.1007/s11227-021-03730-7. URL `https://doi.org/10.1007/s11227-021-03730-7`.

[108] Hongzhang Shan, Katie Antypas, and John Shalf. Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08. IEEE Press, 2008. ISBN 9781424428359.

[109] Timothy J. Sheehan, Allen D. Malony, and Sameer Shende. A Runtime Monitoring Framework for the TAU Profiling System. In *Proceedings of the Third International Symposium on Computing in Object-Oriented Parallel Environments*, ISCOPE '99, page 170–181, Berlin, Heidelberg, 1999. Springer-Verlag. ISBN 3540668187. doi: 10.1007/10704054_18.

[110] Sameer S. Shende, Allen D. Malony, Wyatt Spear, and Karen Schuchardt. Characterizing i/o performance using the tau performance system. In *International Conference on Parallel Computing*, 2011. URL `https://api.semanticscholar.org/CorpusID:361834`.

[111] Shane Snyder, Philip Carns, Robert Latham, Misbah Mubarak, Robert Ross, Christopher Carothers, Babak Behzad, Huong Vu Thanh Luu, Surendra Byna, and Prabhat. Techniques for modeling large-scale hpc i/o workloads. PMBS '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340090. doi: 10.1145/2832087.2832091. URL https://doi.org/10.1145/2832087.2832091.

[112] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K. Lockwood, and Nicholas J. Wright. Modular hpc i/o characterization with darshan. In *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, pages 9–17, 2016. doi: 10.1109/ESPT.2016.006.

[113] Jan Stender, Björn Kolbeck, Felix Hupfeld, Eugenio Cesario, Erich Focht, Matthias Hess, Jesus Malo, and Jonathan Martí. Striping without sacrifices: Maintaining posix semantics in a parallel file system. 01 2008.

[114] Endre Bakken Stovner and Pål Sætrom. PyRanges: Efficient Comparison of Genomic Intervals in Python. *Bioinformatics*, 36(3):918–919, 08 2019. ISSN 1367-4803. doi: 10.1093/bioinformatics/btz615.

[115] Jingwei Sun, Guangzhong Sun, Shiyan Zhan, Jiepeng Zhang, and Yong Chen. Automated performance modeling of hpc applications using machine learning. *IEEE Transactions on Computers*, 69(5):749–763, 2020. doi: 10.1109/TC.2020.2964767.

[116] Hanul Sung, Jiwoo Bang, Alexander Sim, Kesheng Wu, and Hyeonsang Eom. Understanding Parallel I/O Performance Trends Under Various HPC Configurations. In *SNTA'19*, page 29–36, New York, NY, USA, 2019. ACM. ISBN 9781450367615. doi: 10.1145/3322798.3329258.

[117] Houjun Tang, Quincey Koziol, Suren Byna, John Mainzer, and Tonglin Li. Enabling Transparent Asynchronous I/O using Background Threads. In *2019 IEEE/ACM Fourth Int. Parallel Data Systems Workshop (PDSW)*, pages 11–19, 2019. doi: 10.1109/PDSW49588.2019.00006.

[118] Houjun Tang, Quincey Koziol, John Ravi, and Suren Byna. Transparent Asynchronous Parallel I/O Using Background Threads. *IEEE TPDS*, 33(4): 891–902, 2022. doi: 10.1109/TPDS.2021.3090322.

[119] Neda Tavakoli, Dong Dai, and Yong Chen. Log-Assisted Straggler-Aware I/O Scheduler for High-End Computing. In *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, pages 181–189, 2016. doi: 10.1109/ICPPW.2016.38.

[120] 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*, pages 182–189, 1999. doi: 10.1109/FMPC.1999.750599.

[121] Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing noncontiguous accesses in mpi–io. *Parallel Computing*, 28(1):83–105, 2002. ISSN 0167-8191. doi: https://doi.org/10.1016/S0167-8191(01)00129-6. URL https://www.sciencedirect.com/science/article/pii/S0167819101001296.

[122] The HDF Group. Parallel hdf5. https://support.hdfgroup.org/documentation/hdf5-docs/hdf5_topics/ParallelHDF5.html, 2025. Accessed: 2025-02-20.

[123] The pandas Development Team. pandas-dev/pandas: Pandas, feb 2020. URL https://doi.org/10.5281/zenodo.3509134.

[124] Abdul Jabbar Saeed Tipu, Padraig Ó Conbhuí, and Enda Howley. Applying neural networks to predict hpc-i/o bandwidth over seismic data on lustre file system for exseisdat. 25(4):2661–2682, August 2022. ISSN 1386-7857. doi: 10.1007/s10586-021-03347-8. URL https://doi.org/10.1007/s10586-021-03347-8.

[125] Abdul Jabbar Saeed Tipu, Pádraig Ó Conbhuí, and Enda Howley. Artificial neural networks based predictions towards the auto-tuning and optimization of parallel IO bandwidth in HPC system. *Cluster Comput.*, 27(1):71–90, February 2024.

[126] University of Alaska Fairbanks Research Computing Systems. *CENTER1 Lustre Performance / Striping Guide*, 2023. URL https://uaf-rcs.gitbook.io/uaf-rcs-storage-docs/lustre_striping_guide.

[127] Sudharshan S. Vazhkudai, Ross Miller, Devesh Tiwari, Christopher Zimmer, Feiyi Wang, Sarp Oral, Raghul Gunasekaran, and Deryl Steinert. Guide: A scalable information directory service to collect, federate, and analyze logs for operational insights into a leadership hpc facility. In *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[128] Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C. Roth. Scalable i/o tracing and analysis. PDSW '09, page 26–31, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605588834. doi: 10.1145/1713072.1713080. URL https://doi.org/10.1145/1713072.1713080.

[129] 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)*, pages 1–11, Los Alamitos, CA, USA, may 2009. IEEE Computer Society. doi: 10.1109/IPDPS.2009.5161029.

[130] Lipeng Wan, Matthew Wolf, Feiyi Wang, Jong Youl Choi, George Ostrouchov, and Scott Klasky. Comprehensive measurement and analysis of the user-perceived i/o performance in a production leadership-class storage system. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1022–1031, 2017. doi: 10.1109/ICDCS.2017.257.

[131] Chen Wang, Jinghan Sun, Marc Snir, Kathryn Mohror, and Elsa Gonsiorowski. Recorder 2.0: Efficient parallel i/o tracing and analysis. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1–8, 2020. doi: 10.1109/IPDPSW50202.2020.00176.

[132] Chen Wang, Kathryn Mohror, and Marc Snir. File system semantics requirements of HPC applications. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, pages 19–30, 2021.

[133] Chen Wang, Izzet Yildirim, Hariharan Devarajan, Kathryn Mohror, and Marc Snir. Recorder: Comprehensive parallel i/o tracing and analysis, 2025. URL `https://arxiv.org/abs/2501.04654`.

[134] Teng Wang, Shane Snyder, Glenn Lockwood, Philip Carns, Nicholas Wright, and Suren Byna. IOMiner: Large-Scale Analytics Framework for Gaining Knowledge from I/O Logs. In *CLUSTER*, pages 466–476, 2018. doi: 10.1109/CLUSTER.2018.00062.

[135] Teng Wang, Suren Byna, Glenn K. Lockwood, Shane Snyder, Philip Carns, Sunggon Kim, and Nicholas J. Wright. A zoom-in analysis of i/o logs to detect root causes of i/o performance bottlenecks. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 102–111, 2019. doi: 10.1109/CCGRID.2019.00021.

[136] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. pages 17–33, 01 2008.

[137] D. C. Wells, E. W. Greisen, and R. H. Harten. FITS - a Flexible Image Transport System. *Astronomy and Astrophysics Supplement Series*, 44:363, June 1981.

[138] Leland Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag, Berlin, Heidelberg, 2005. ISBN 0387245448.

[139] B. Wolman and T. M. Olson. Iobench: a system independent io benchmark. *SIGARCH Comput. Archit. News*, 17(5):55–70, sep 1989. ISSN 0163-5964. doi: 10.1145/71302.71309. URL `https://doi.org/10.1145/71302.71309`.

[140] Nicholas J. Wright, Wayne Pfeiffer, and Allan Snavely. Characterizing parallel scaling of scientific applications using ipm. 2009. URL `https://api.semanticscholar.org/CorpusID:16489254`.

[141] Steven Wright, S.D. Hammond, John Pennycook, R.F. Bird, Andy Herdman, I. Miller, A. Vadgama, Abhir Bhalerao, and Stephen Jarvis. Parallel file system analysis through application i/o tracing. *The Computer Journal*, 56:141–155, 02 2013. doi: 10.1093/comjnl/bxs044.

[142] Bing Xie, Houjun Tang, Suren Byna, Jesse Hanley, Quincey Koziol, Tonglin Li, and Sarp Oral. Battle of the defaults: Extracting performance characteristics of hdf5 under production load. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 51–60, 2021. doi: 10.1109/CCGrid51090.2021.00015.

[143] Cong Xu, Shane Snyder, Omkar Kulkarni, Vishwanath Venkatesan, Phillip Carns, Surendra Byna, Robert Sisneros, and Kalyana Chadalavada. DXT: Darshan eXtended Tracing. 1 2019.

[144] Yiheng Xu, Pranav Sivaraman, Hariharan Devarajan, Kathryn M. Mohror, and Abhinav Bhatele. Ml-based modeling to predict I/O performance on different storage sub-systems. *CoRR*, abs/2312.06131, 2023. doi: 10.48550/ARXIV.2312.06131. URL `https://doi.org/10.48550/arXiv.2312.06131`.

[145] Bin Yang, Xu Ji, Xiaosong Ma, Xiyang Wang, Tianyu Zhang, Xiupeng Zhu, Nosayba El-Sayed, Haidong Lan, Yibo Yang, Jidong Zhai, Weiguo Liu, and Wei Xue. End-to-end I/O monitoring on a leading supercomputer. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 379–394, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL `https://www.usenix.org/conference/nsdi19/presentation/yang`.

[146] Orcun Yildiz, Matthieu Dorier, Shadi Ibrahim, Robert B. Ross, and Gabriel Antoniu. On the root causes of cross-application i/o interference in hpc storage systems. *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 750–759, 2016. URL `https://api.semanticscholar.org/CorpusID:17570971`.

[147] Jie Yu, Guangming Liu, Wenrui Dong, Xiaoyong Li, Jian Zhang, and Fuxing Sun. On the Load Imbalance Problem of I/O Forwarding Layer in HPC Systems. In *ICCC*, pages 2424–2428, 2017. doi: 10.1109/CompComm.2017.8322970.

[148] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, Max P. Katz, Andrew Myers, Tan Nguyen, Andrew Nonaka, Michele Rosso, Samuel Williams, and Michael Zingale. AMReX: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software*, 4(37):1370, May 2019. doi: 10.21105/joss.01370.

[149] Weiqun Zhang et al. AMReX: Block-structured adaptive mesh refinement for multiphysics applications. *The International Journal of High Performance Computing Applications*, 35(6):508–526, 2021. doi: 10.1177/10943420211022811.

[150] Tiezhu Zhao, Verdi March, Shoubin Dong, and Simon See. Evaluation of a performance model of lustre file system. In *2010 Fifth Annual ChinaGrid Conference*, pages 191–196, 2010. doi: 10.1109/ChinaGrid.2010.38.