SmartIO: A Lightweight End-to-End Workflow for Runtime I/O Optimization of HPC Systems

Hammad Bin Ather
University of Oregon
Eugene, OR, USA
hather@uoregon.edu

Chen Wang*
Nanyang Technological University
Singapore, Singapore
chen.wang@ntu.edu.sg

Hariharan Devarajan
Lawrence Livermore National Laboratory
Livermore, CA, USA
hariharandev1@llnl.gov

Hank Childs
University of Oregon
Eugene, OR, USA
hank@uoregon.edu

Kathryn Mohror
Lawrence Livermore National Laboratory
Livermore, CA, USA
mohror1@llnl.gov

Abstract

The I/O subsystem is often a critical bottleneck in HPC systems, making its optimization crucial for performance. Existing efforts to optimize the I/O performance of HPC applications either rely on offline tuning, exhaustive parameter searches, or extensive training of machine learning models-approaches that are time consuming and resource intensive. This work introduces *SmartIO*, an end-to-end workflow to optimize the I/O performance of HPC systems at runtime without requiring prior model training, profiling, or parameter searches. *SmartIO* leverages context-free grammars to predict future I/O calls during the application runtime, learns key I/O characteristics from predicted calls, and proactively optimizes performance before those calls occur using a rules-based mapping engine. The rules-based mapping engine, constructed through literature review and empirical evaluation, serves as a translation mechanism for insights into optimizations at runtime. Our evaluation shows that *SmartIO* achieves up to ~ 13× and ~ 12× improvements in IOR read and write bandwidth, respectively, and delivers a ~ 4× speedup in overall I/O bandwidth for Flash-X—all with negligible runtime overhead. Compared to the state-of-the-art I/O optimization tools, *SmartIO* delivers comparable or better performance while drastically reducing the tuning cost.

CCS Concepts

• **Information systems** → **Hierarchical storage management**; • **Computing methodologies** → *Parallel computing methodologies.*

Keywords

I/O Performance Tuning, I/O Prediction, Insights

*The majority of this work was carried out while Chen Wang was at Lawrence Livermore National Laboratory.

SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/25/11
https://doi.org/10.1145/3731599.3767509

ACM Reference Format:
Hammad Bin Ather, Chen Wang, Hariharan Devarajan, Hank Childs, and Kathryn Mohror. 2025. SmartIO: A Lightweight End-to-End Workflow for Runtime I/O Optimization of HPC Systems. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3731599.3767509

1 Introduction

High-performance computing (HPC) systems run critical workloads with exceptional speed and efficiency, leveraging state-of-the-art hardware and computing resources. These systems are made up of thousands of interconnected nodes, each made of various system components such as 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 [14], with various underlying causes, including unbalanced I/O workload [38], I/O resource contention [42], and other factors [28, 35].

Significant efforts have been made to optimize the I/O performance of HPC systems. However, these efforts suffer from high resource and time overhead. Optimization frameworks like Drishti IO [2, 7] analyze profiling data to detect I/O bottlenecks in the application and provide actionable recommendations to the user. However, Drishti IO operates offline and the optimizations can only be applied manually in subsequent runs, which is impractical for long-running and large-scale applications. Autotuning approaches [4–6] 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 (ML)-based I/O prediction and optimization [19, 32, 40] also face overhead challenges, as it relies on extensive training data collected from multiple simulation runs to train the ML models, making this approach time- and resource-intensive. Furthermore, ML-based tools often lack generality, as they are trained to optimize specific applications and libraries only. Collectively, these approaches provide benefit in optimizing I/O performance, but require a large overhead cost. The main premise behind our research is that such overhead costs serve as a barrier to adoption,

and that new research is needed on approaches that can provide performance optimization with minimal overhead.

This paper introduces *SmartIO*, which combines three components together into an end-to-end fully runtime workflow: (1) prediction of future I/O calls through detection of recurring patterns in I/O using context-free grammars (CFG) [9], (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 can dynamically optimize parameters of HDF5, ROMIO, and Lustre parallel file system (PFS) at runtime, without requiring prior profiling, model training, or exhaustive parameter searches. Our evaluations of *SmartIO* show approximately a 13× improvement in read bandwidth and a 12× improvement in write bandwidth for IOR [27], along with a ∼ 4× increase in overall I/O bandwidth for Flash-X [13]. These gains are achieved with minimal runtime overhead—less than 1 seconds for IOR and ∼ 4 seconds for Flash-X. Additionally, our comparison with the state-of-the-art I/O optimization tools shows that *SmartIO* achieves comparable and, in some cases, better performance than the state-of-the-art, while cutting down significantly on the tuning overhead. Finally, as the driving premise for *SmartIO's* approach is lowering barriers for user adoption, it is important that nothing about *SmartIO's* prevents such adoption; to demonstrate its viability in practice, *SmartIO* has been integrated into the Recorder I/O tracing framework [36, 37], enabling out-of-the-box use without requiring application code modifications or recompilation.

In all, this work makes the following key contributions:

(1) A novel end-to-end workflow designed to predict and optimize I/O performance at runtime without reliance on prior model training, profiling, or parameter searches.
(2) A rules-based mapping engine, constructed through literature review and empirical evaluation, to translate 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) Demonstration of the effectiveness of this workflow on an HPC benchmark and a real-world application and a comparative study with the state-of-the-art.

2 Related Work

This section reviews existing approaches for optimizing parallel I/O, as well as methods for pattern detection and prediction of future I/O calls. We discuss their limitations and explain how *SmartIO* overcomes these challenges.

Offline I/O Tuning. Drishti IO [2, 3, 7] is a command line tool that guides end-users in optimizing the I/O performance of their applications by detecting common performance pitfalls and providing actionable recommendations using a heuristic-based approach. TOKIO [41] and UMAMI [18] provide a holistic view of the I/O performance by integrating data from multiple HPC monitoring and profiling tools.

Autotuning. This work [6] introduces a genetic algorithm-based auto-tuning approach to find the optimal parameter configurations for the given application. The follow-up work [5] uses empirical models of the I/O performance to reduce the overhead of running the genetic algorithm introduced in [6]. Another study [4] presents a machine learning–driven autotuning approach that optimizes the collective I/O operations of ROMIO using random forest regression.

Pattern Detection and Prediction. Omnisc'IO [12] constructs a context-free grammar-based model to predict when and where future I/O operations will occur and how much data will be accessed. The context-free grammar can capture both spatial and temporal access patterns, enabling accurate forecasts of future I/O operations.

All the aforementioned tools have some limitations that *SmartIO* addresses. Offline I/O optimization tools rely on insights from previous runs, which may not be practical for large-scale applications. ML- and auto-tuning–based methods demand extensive training data and tuning time, creating high overhead. Omnisc'IO instruments only at the POSIX and MPI-IO layers, restricting access to higher-level optimizations (e.g., alignment, data transfer mode). It also provides limited access to function parameters essential for deriving access pattern information, like offset, size of communication, spatial locality, etc. Additionally, Omnisc'IO maintains separate grammars for size and offset, each storing only 24 symbols, and predictions are made independently across these grammars.

SmartIO, with its end-to-end workflow, requires no model training, profiling, or exhaustive parameter searches, enabling real-time I/O optimization with minimal time and resource overhead. Moreover, the grammar-based model provided by Recorder, which *SmartIO* builds on for its prediction capabilities, intercepts a broader range of I/O calls across multiple layers, including POSIX, MPI-IO, and high-level libraries like HDF5, without requiring additional implementation efforts. This comprehensive tracing captures all function parameters, enabling detailed analysis of I/O patterns and library-specific tunings.

3 SmartIO

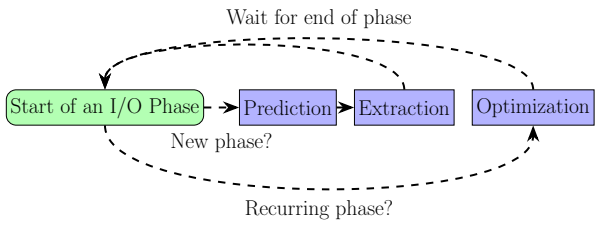

Figure 1: Overview of the *SmartIO's* workflow: *SmartIO* invokes the prediction and extraction component for a new I/O phase and optimization component for a recurring I/O phase. All components operate dynamically at runtime.

SmartIO comprises three components: (1) **prediction**, (2) **extraction**, and (3) **optimization**. Prediction detects recurring I/O patterns, extraction derives insights from predicted calls, and optimization applies performance optimizations via a rules-based engine. These components are invoked dynamically at runtime, depending on the I/O phase encountered.

HPC applications often feature recurring I/O phases, such as checkpointing and data loading [8, 11, 34]. *SmartIO* detects the start of an I/O phase, learns its behavior through prediction and extraction, and applies optimizations during subsequent recurring

phases. At runtime, it classifies new vs. recurring phases to determine which components to invoke, as shown in Fig. 1. To identify the start of an I/O phase, *SmartIO* monitors runtime calls from HDF5, MPI-IO, and POSIX interfaces. We found that `H5Fcreate` consistently marks the beginning of each I/O phase, unlike other calls such as `MPI_File_open`, which occur multiple times within a single phase. *SmartIO* then uses the file name from `H5Fcreate` to classify I/O phases. If the file (e.g., checkpoint or plot) has been encountered before, the phase is recurring and reuses previously extracted insights for optimization. Otherwise, prediction and extraction are re-triggered to profile the new phase. Optimizations are applied via HDF5, ROMIO, and Lustre interfaces. HDF5 settings (alignment, cache, and transfer mode) are adjusted immediately using parameters from `H5Fcreate`/`H5open`. ROMIO and Lustre require deferred tuning: `H5Pset_fapl_mpio` and Lustre striping must be modified during the next encounter, as their control points precede `H5Fcreate` in the current phase.

3.1 First Component: Prediction

This component uses Recorder's CFG-based pattern recognition to predict future I/O calls from the current execution point.

3.1.1 Pattern-Recognition: Recorder constructs per-process context-free grammars (CFGs) using the Sequitur algorithm [22] to identify recurring I/O patterns. Each terminal symbol in the CFG corresponds to a unique I/O call signature stored in a call signature table (CST), implemented as a hash table. When a new I/O call is intercepted, it is either mapped to an existing symbol or assigned a new one, updating both the CFG and CST accordingly.

3.1.2 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 checks if this symbol can be used to predict future I/O calls. It searches for CFG rules (excluding the start rule *S*) that begin with the current symbol and returns all matching candidates, resulting in multiple predictions. To resolve multiple predictions, the algorithm uses a hashtable to rank candidate rules based on their frequency of occurrence in the start rule *S*, selecting the most frequently occurring rule. Once a rule has been chosen as the final prediction from the hashtable, the algorithm recursively expands any non-terminal symbols in the predicted rule to extract a full stack trace of the predicted I/O calls. 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.

3.2 Second Component: Extraction

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 I/O, the component detects specific MPI-IO function calls in the stack, such as `MPI_File_write_at_all` and `MPI_File_read_at_all`.

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 1 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.

3.3 Third Component: Optimization

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 an 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. The 10 rules governing these optimizations and their supporting evidence from literature and empirical evaluation are presented in Table 2. The following subsections present empirical evaluations for selected rules.

3.3.1 Evaluations for HDF5 Data Transfer Mode Rules: Table 2 presents established rules for shared file configurations, but no prior work evaluated data transfer modes in file-per-process setups. To address this, we ran IOR with 32 nodes, 1024 processes, 512MB blocks, and 64KB transfers, comparing collective and independent I/O. Results showed only a 3.17% performance difference, favoring independent I/O. Given the minimal impact, we leave the transfer mode unchanged for file-per-process configurations.

3.3.2 Evaluations for ROMIO Rules: Due to limited prior work, we empirically derived ROMIO optimization rules using IOR with 8 nodes, 400 processes, 128MB blocks, and 4MB transfers. Increasing aggregators to 8 per node improved performance. Setting *cb_buffer_size* to 2MB had minimal impact, so the default is retained. Disabling *romio_cb_write* reduced execution time by 76%, likely by avoiding collective buffering overhead[17]. However, this result applies to sequential shared-file accesses—prior work [29] suggests keeping the default for non-sequential patterns. Data sieving parameters showed negligible performance differences, so defaults are preserved.

3.3.3 Evaluations for Lustre Stripe Count Rules: As shown in Table 2, prior work provides guidance for selecting stripe counts, but the method of applying striping—static vs. progressive—can significantly affect performance. Since only one study [24] evaluates both, we conducted a scaling experiment using IOR with block sizes of 32MB to 256MB, a fixed 4MB transfer size, 32 compute nodes, and 1024 processes. Static striping used the maximum stripe count (−1),

Table 1: List of the function calls across 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	✗

Table 2: 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 [10, 17]	✗
	2. Use collective I/O for random or non-sequential reads/writes to a shared file [10, 20, 29–31]	✗
	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 [39]	✗
HDF5 Metadata Cache	**5.** Set metadata cache size between 1-16MB if transfer size < 16MB, otherwise set it >= 16MB [31]	✗
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 [17]	✓
ROMIO Data Sieving	**8.** Keep default values for the data sieving parameters [29]	✓
Lustre Stripe Count	**9.** For file-per-process configurations, set stripe count to 1 [1, 16, 21, 23, 33]	✗
	10. For a shared file, set a progressive stripe layout if Lustre version > 2.10; otherwise, set stripe count according to file size [1, 16, 21, 23, 25, 26, 33]	✓

while progressive striping applied a layout that increased the stripe count with file size (e.g., 1 stripe for ≤ 256MB, 4 for ≤ 4GB, max beyond). Our results showed that progressive striping consistently outperformed static, especially as file size grew. While the precise cause is unclear, adaptive striping behavior likely helps maintain balanced I/O. These findings support results in [24].

4 Evaluation

This section showcases the effectiveness of *SmartIO* using a benchmark, IOR [27], and a real-world application, Flash-X [13]. 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).

4.1 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 3.

Fig. 2 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

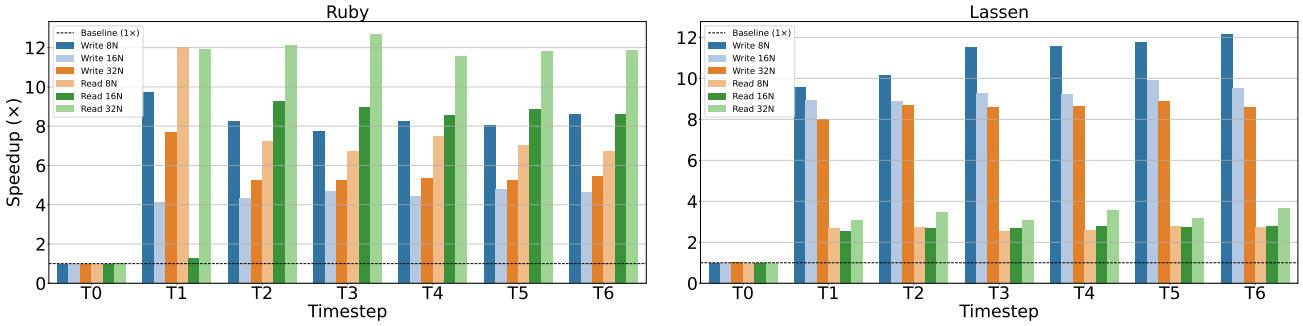

Figure 2: Real-time read and write bandwidth speedup relative to the baseline at each timestep on Ruby and Lassen using *SmartIO* on IOR with 8, 16, and 32 nodes (256, 512, and 1024 MPI processes, respectively).

Table 3: Mapping of insights to Table 2 rules and applied optimizations before timesteps 1 and 2.

Layer	Insight → Rule	Optimization
HDF5	2 → 4	**(1)** Changed from no **alignment** to 1MB
	2 → 5	**(2)** Changed 2MB **metadata cache size** to 8MB
	1, 3, 4, 5 → 1	**(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

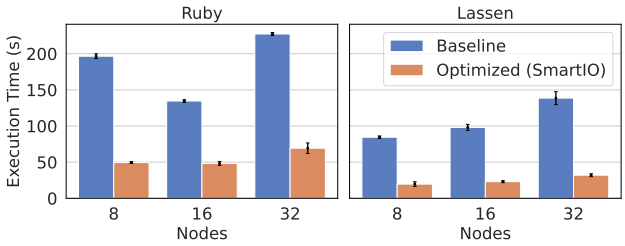

Figure 3: Comparison of baseline and optimized IOR execution time on Ruby and Lassen with 8, 16, and 32 nodes.

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.

Fig. 3 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.

4.2 Flash-X

Flash-X is an open-source, highly composable software system that can simulate various physical phenomena across various scientific domains [13]. 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 4.

(1) **Data Transfer Mode:** Future calls will utilize both collective and independent I/O
(2) **Transfer Size:** Each transfer size will be ≤ 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

Fig. 4 presents the real-time performance speedup relative to the baseline gained by *SmartIO* on Flash-X Sedov. As can be seen

Table 4: Mapping of insights to Table 2 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

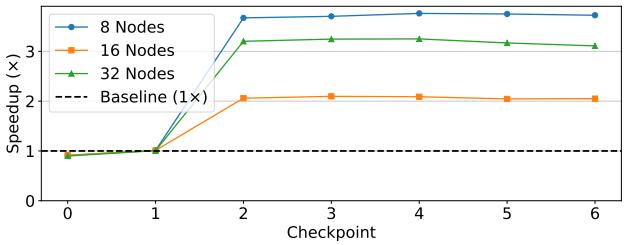

Figure 4: Real-time I/O bandwidth speedup relative to the baseline at each checkpoint on Ruby using *SmartIO* on Flash-X with 8, 16, and 32 nodes (448, 896, and 1792 MPI processes, respectively).

from the results, *SmartIO* delivered a consistent bandwidth speedup even as the scale and complexity of the application increased. The maximum speedup of approximately ~ 4× is achieved on 8 nodes and remains consistent across all I/O phases (checkpoints). On 16 nodes, the speedup decreased to around 2×, but increased again to over 3× on 32 nodes.

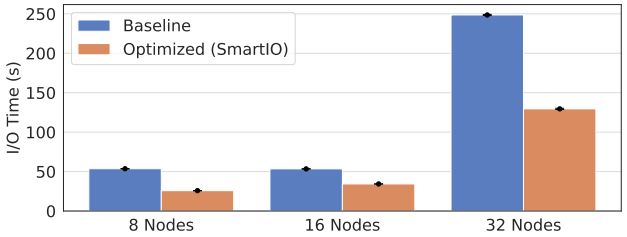

Figure 5: Comparison of baseline and optimized Flash-X I/O time on Ruby with 8, 16, and 32 nodes.

Fig. 5 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.

4.3 Overhead

SmartIO incurs minimal runtime overhead, primarily from its prediction and extraction components, while the optimization component contributes negligibly. Fig. 6 shows overhead results from large-scale runs on IOR and Flash-X. For IOR (32 nodes, 1024 processes), the total overhead was 0.89s (1.26% of optimized runtime),

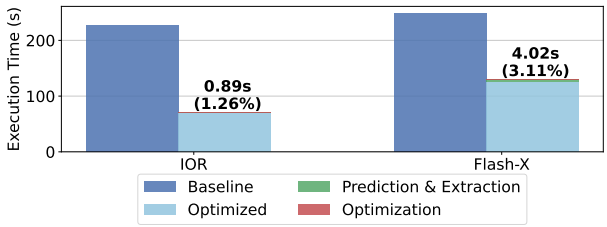

Figure 6: Overhead breakdown of the three components of *SmartIO*: prediction, extraction, and optimization

with 0.88s from prediction/extraction and just 0.006s from optimization. Despite this low overhead, *SmartIO* reduced total execution time by 69.5%. For Flash-X (32 nodes, 1792 processes), overhead rose to 4.02s (3.11%), again mostly due to prediction/extraction (4s), with negligible cost from optimization (0.022s). This higher cost reflects Flash-X's complex I/O patterns. Even so, *SmartIO* achieved a 48% reduction in I/O time.

4.4 Comparison with the state-of-the-art

We compare *SmartIO* 's speedup and tuning overhead with state-of-the-art tools: Drishti IO [2, 7] and an autotuning framework [5, 6]. As discussed in Section 2, Drishti IO is an offline analyzer that recommends optimizations based on trace analysis, while the autotuning framework uses genetic algorithms and performance modeling to tune I/O parameters.

Table 5: Comparison of achieved speedup and tuning overhead on the IOR benchmark: *SmartIO* vs. state-of-the-art

Tuning Tool	Speedup (×)	Tuning Overhead (s)
Drishti IO [2, 7]	1	37.59
Autotuning Framework [5, 6]	9	36000
SmartIO	6	0.08

For the autotuning framework, we use the reported results in [5], which evaluated IOR with 8 segments, write-only access, and 32MB block/transfer size on 512 cores. To match their setup, we ran *SmartIO* with the same IOR configuration on Lustre. Table 5 summarizes the results: *SmartIO* achieved a 6× speedup with only 0.08s of tuning overhead, while the autotuning framework achieved 9× but required over 10 hours—more than 450,000× the overhead. Flash-X was not evaluated in the autotuning study.

We also ran Drishti IO under the same IOR setup. As an offline tool, it required two runs: one for collecting logs and another for applying its recommendations, incurring 37.59s of overhead. However, it showed no performance gains on our test system, though its effectiveness may vary across platforms and configurations.

5 Discussion and Future Work

While effective, *SmartIO* 's rule-based design has a key limitation: its static rules do not adapt to system-specific variations. Though these rules are grounded in literature and empirical evaluation, they remain fixed across systems. However, this design also provides a major benefit—*SmartIO* avoids performance modeling, profiling,

or parameter search, incurring no tuning overhead, while also delivering substantial speedup as shown in Section 4.

Future work includes adding a feedback loop to detect and respond to runtime performance degradation and extending support to other I/O libraries, such as PnetCDF [15].

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under the DOE Early Career Research Program (LLNL-CONF-2004393).

References

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

[2] Hammad Ather, Jean Luca Bez, Boyana Norris, and Suren Byna. 2023. Illuminating the I/O Optimization Path of Scientific Applications. In *High Performance Computing*, Abhinav Bhatele, Jeff Hammond, Marc Baboulin, and Carola Kruse (Eds.). Springer Nature Switzerland, Cham, 22–41.

[3] Hammad Ather, Jean Luca Bez, Yankun Xia, and Suren Byna. 2024. Drilling Down I/O Bottlenecks with Cross-layer I/O Profile Exploration. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 532–543. doi:10.1109/IPDPS57955.2024.00053

[4] Ayşe Bağbaba. 2020. Improving Collective I/O Performance with Machine Learning Supported Auto-tuning. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 814–821. doi:10.1109/IPDPSW50202.2020.00138

[5] Babak Behzad, Surendra Byna, Stefan M. Wild, Mr. Prabhat, and Marc Snir. 2014. Improving parallel I/O autotuning with performance modeling. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing* (Vancouver, BC, Canada) *(HPDC '14)*. Association for Computing Machinery, New York, NY, USA, 253–256. doi:10.1145/2600212.2600708

[6] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Prabhat, Ruth Aydt, Quincey Koziol, and Marc Snir. 2013. Taming parallel I/O complexity with auto-tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 68, 12 pages. doi:10.1145/2503210.2503278

[7] Jean Luca Bez, Hammad Ather, and Suren Byna. 2022. Drishti: Guiding End-Users in the I/O Optimization Journey. In *2022 IEEE/ACM International Parallel Data Systems Workshop (PDSW)*. 1–6. doi:10.1109/PDSW56643.2022.00006

[8] Jesus Carretero, Emmanuel Jeannot, Guillaume Pallez, David E. Singh, and Nicolas Vidal. 2020. Mapping and scheduling HPC applications for optimizing I/O. In *Proceedings of the 34th ACM International Conference on Supercomputing* (Barcelona, Spain) *(ICS '20)*. Association for Computing Machinery, New York, NY, USA, Article 33, 12 pages. doi:10.1145/3392717.3392764

[9] William Y.C. Chen. 1993. Context-free grammars, differential operators and formal power series. *Theoretical Computer Science* 117, 1 (1993), 113–129. doi:10.1016/0304-3975(93)90307-F

[10] Christian M. Chilan and Kent Yang. 2006. *Performance Comparison of Collective I/O and Independent I/O with Derived Datatypes.* Technical Report. The HDF Group. https://support.hdfgroup.org/releases/hdf5/documentation/rfc/coll_ind_dd6.pdf

[11] Hariharan Devarajan and Kathryn Mohror. 2022. Extracting and characterizing I/O behavior of HPC workloads. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. 243–255. doi:10.1109/CLUSTER51413.2022.00037

[12] Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, and Rob Ross. 2014. 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*. 623–634. doi:10.1109/SC.2014.56

[13] 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. 2022. Flash-X: A multiphysics simulation software instrument. *SoftwareX* 19 (2022), 101168. doi:10.1016/j.softx.2022.101168

[14] Peter Harrington, Wucherl Yoo, Alexander Sim, and Kesheng Wu. 2017. 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)*. 4.

[15] Robert Latham. 2011. *NetCDF I/O Library, Parallel.* Springer US, Boston, MA, 1283–1291. doi:10.1007/978-0-387-09766-4_235

[16] Si Liu, Lei Huang, Hang Liu, Amit Ruhela, Virginia Trueheart, Susan Lindsey, and Quan Yuan. 2021. 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* (Boston, MA, USA) *(PEARC '21)*. Association for Computing Machinery, New York, NY, USA, Article 5, 8 pages. doi:10.1145/3437359.3465570

[17] Wei Liu, Kai Wu, Jialin Liu, Feng Chen, and Dong Li. 2017. Performance Evaluation and Modeling of HPC I/O on Non-Volatile Memory. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*. 1–10. doi:10.1109/NAS.2017.8026869

[18] Glenn K. Lockwood, Wucherl Yoo, Suren Byna, Nicholas J. Wright, Shane Snyder, Kevin Harms, Zachary Nault, and Philip Carns. 2017. 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* (Denver, Colorado) *(PDSW-DISCS '17)*. Association for Computing Machinery, New York, NY, USA, 55–60. doi:10.1145/3149393.3149395

[19] Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert Latham, Robert Ross, Shane Snyder, and Stefan M. Wild. 2018. Machine Learning Based Parallel I/O Predictive Modeling: A Case Study on Lustre File Systems. In *High Performance Computing*, Rio Yokota, Michèle Weiland, David Keyes, and Carsten Trinitis (Eds.). Springer International Publishing, Cham, 184–204.

[20] Sandra Mendez, Sebastian Lührs, Dominic Sloan-Murphy, Andrew Turner, and Volker Weinberg. 2019. Best Practice Guide - Parallel I/O. https://prace-ri.eu/wp-content/uploads/Best-Practice-Guide_Parallel-IO.pdf

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

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

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

[24] Joel Reed, Jeremy Archuleta, Michael Brim, and Josh Lothian. 2015. Evaluating Dynamic File Striping For Lustre.

[25] Subhash Saini, Jason Rappleye, Johnny Chang, David Barker, Piyush Mehrotra, and Rupak Biswas. 2012. I/O performance characterization of Lustre and NASA applications on Pleiades. In *2012 19th International Conference on High Performance Computing*. 1–10. doi:10.1109/HiPC.2012.6507507

[26] Marco Seiz, Philipp Offenhäuser, Stefan Andersson, Johannes Hötzer, Henrik Hierl, Britta Nestler, and Michael Resch. 2021. Lustre I/O performance investigations on Hazel Hen: experiments and heuristics. *J. Supercomput.* 77, 11 (Nov. 2021), 12508–12536. doi:10.1007/s11227-021-03730-7

[27] Hongzhang Shan, Katie Antypas, and John Shalf. 2008. 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* (Austin, Texas) *(SC '08)*. IEEE Press, Article 42, 12 pages.

[28] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K. Lockwood, and Nicholas J. Wright. 2016. Modular HPC I/O Characterization with Darshan. In *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*. 9–17. doi:10.1109/ESPT.2016.006

[29] R. Thakur, W. Gropp, and E. Lusk. 1999. Data sieving and collective I/O in ROMIO. In *Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation*. 182–189. doi:10.1109/FMPC.1999.750599

[30] Rajeev Thakur, William Gropp, and Ewing Lusk. 2002. Optimizing noncontiguous accesses in MPI–IO. *Parallel Comput.* 28, 1 (2002), 83–105. doi:10.1016/S0167-8191(01)00129-6

[31] The HDF Group. 2025. Parallel HDF5. https://support.hdfgroup.org/documentation/hdf5-docs/hdf5_topics/ParallelHDF5.html. Accessed: 2025-02-20.

[32] Abdul Jabbar Saeed Tipu, Padraig Ó Conbhuí, and Enda Howley. 2022. Applying neural networks to predict HPC-I/O bandwidth over seismic data on lustre file system for ExSeisDat. 25, 4 (Aug. 2022), 2661–2682. doi:10.1007/s10586-021-03347-8

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

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

[35] Chen Wang, Kathryn Mohror, and Marc Snir. 2024. Formal definitions and performance comparison of consistency models for parallel file systems. *IEEE*

Transactions on Parallel and Distributed Systems 35, 6 (2024), 1092–1106.

[36] Chen Wang, Jinghan Sun, Marc Snir, Kathryn Mohror, and Elsa Gonsiorowski. 2020. Recorder 2.0: Efficient Parallel I/O Tracing and Analysis. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1–8. doi:10.1109/IPDPSW50202.2020.00176

[37] Chen Wang, Izzet Yildirim, Hariharan Devarajan, Kathryn Mohror, and Marc Snir. 2025. Recorder: Comprehensive Parallel I/O Tracing and Analysis. *arXiv preprint arXiv:2501.04654* (2025).

[38] Teng Wang, Suren Byna, Glenn K. Lockwood, Shane Snyder, Philip Carns, Sunggon Kim, and Nicholas J. Wright. 2019. 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)*. 102–111. doi:10.1109/CCGRID.2019.00021

[39] Bing Xie, Houjun Tang, Suren Byna, Jesse Hanley, Quincey Koziol, Tonglin Li, and Sarp Oral. 2021. 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)*. 51–60. doi:10.1109/CCGrid51090.2021.00015

[40] Yiheng Xu, Pranav Sivaraman, Hariharan Devarajan, Kathryn M. Mohror, and Abhinav Bhatele. 2023. ML-based Modeling to Predict I/O Performance on Different Storage Sub-systems. *CoRR* abs/2312.06131 (2023). doi:10.48550/ARXIV.2312.06131 arXiv:2312.06131

[41] Bin Yang, Wei Xue, Tianyu Zhang, Shichao Liu, Xiaosong Ma, Xiyang Wang, and Weiguo Liu. 2023. End-to-end I/O Monitoring on Leading Supercomputers. *ACM Trans. Storage* 19, 1, Article 3 (Jan. 2023), 35 pages. doi:10.1145/3568425

[42] Orcun Yildiz, Matthieu Dorier, Shadi Ibrahim, Robert B. Ross, and Gabriel Antoniu. 2016. On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems. *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2016), 750–759. https://api.semanticscholar.org/CorpusID:17570971

Appendix: Artifact Description

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

C_1 Introduction of SmartIO, a lightweight end-to-end workflow that predicts and optimizes I/O performance at runtime without prior profiling, parameter searches, or ML training.

C_2 Design of a rules-based mapping engine, constructed from literature review and empirical evaluation, to dynamically tune HDF5, ROMIO, and Lustre settings at runtime.

C_3 Integration of SmartIO into the Recorder I/O tracing framework, enabling transparent adoption without requiring code modifications or recompilation.

C_4 Comprehensive evaluation on IOR and Flash-X across two HPC systems, demonstrating up to 13× read, 12× write, and 4× overall I/O bandwidth speedups with negligible runtime overhead.

A.2 Computational Artifacts

A_1 SmartIO source code and runtime modules integrated with Recorder [Link].

A_2 Experiment scripts, job submission files, and input decks for IOR and Flash-X evaluations [Link].

A_3 Processed datasets and analysis scripts [Link] for reproducing figures and tables [Link].

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_3	Tables I–II, Fig. 1
A_2	C_1, C_4	Tables III–IV, Figs. 2–6
A_3	C_4	Table V, all performance plots

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

Implements the SmartIO runtime workflow (prediction, extraction, optimization) and the rules-based engine. Supports contributions C_1–C_3.

Expected Results

Expected to reproduce runtime optimizations by dynamically tuning HDF5 transfer mode, metadata cache, alignment, ROMIO collective buffering, and Lustre striping. Results should show significant bandwidth improvements on applications with negligible runtime overhead (< 5s).

Expected Reproduction Time (in Minutes)

Setup: 30–40 min (compilation with MPI, HDF5, Recorder). Execution: < 10 min for small-scale IOR runs. Analysis: ~5 min.

Artifact Setup (incl. Inputs)

Hardware. Linux-based HPC system with MPI support. Lustre or GPFS file system recommended for reproducing paper-scale results.

Software.

- Recorder 2.0 https://github.com/uiuc-hpc/Recorder
- MPI (v3.1 or later)
- HDF5 (v1.12 or later)
- GCC (v9.0 or later)

Datasets / Inputs. Not required for functionality. Compatible with any MPI I/O workload.

Installation and Deployment. Standard `make` build system. Requires MPI compiler wrappers (e.g., `mpicc`) and HDF5 development headers.

Artifact Execution

Workflow: T_1: Intercept I/O calls using Recorder. T_2: Generate context-free grammar (CFG) and predict future I/O patterns. T_3: Extract insights and apply runtime optimizations.

Dependencies: $T_1 \rightarrow T_2 \rightarrow T_3$.

Artifact Analysis (incl. Outputs)

Logs of predictions, extracted insights, and applied optimizations. Can be validated against optimization rules (Table II).

B.2 Computational Artifact A_2

Relation To Contributions

Provides reproducible evaluation using IOR and Flash-X. Supports C_1 and C_4.

Expected Results

Reproduces scaling studies demonstrating:

- IOR: 13× read / 12× write speedup, 3–4× execution time reduction.
- Flash-X: 2–4× I/O time reduction across scales.

Expected Reproduction Time (in Minutes)

Setup: ~15 min. Execution: 30–60 min per run (depending on node count). Analysis: 10–15 min.

Artifact Setup (incl. Inputs)

Hardware. HPC system with at least 8 nodes. Larger scales (up to 32 nodes) recommended for full reproduction.

Software.

- IOR v3.3.1 https://github.com/hpc/ior
- Flash-X https://flash-x.org
- SLURM or equivalent job scheduler

Datasets / Inputs. IOR inputs (block size, transfer size) and Flash-X parameter files provided [Link].

Installation and Deployment. IOR: `./configure && make`. Flash-X: standard `make` build with MPI and HDF5 [Link].

Artifact Execution

Workflow: T_1: Run IOR baseline and SmartIO-enabled cases. T_2: Collect logs and Recorder traces. T_3: Run Flash-X Sedov baseline vs. SmartIO. T_4: Parse logs and generate plots.

Dependencies: $T_1 \rightarrow T_2 \rightarrow T_4$; $T_3 \rightarrow T_4$.

Artifact Analysis (incl. Outputs)

Performance results (CSV) and plots equivalent to Figs. 2–6 and Tables III–IV.

B.3 Computational Artifact A_3

Relation To Contributions

Provides pre-processed datasets and plotting scripts for regenerating figures/tables. Supports C_4.

Expected Results

Should reproduce paper figures (performance speedups, overheads, comparisons) without rerunning experiments [Link] and [Link].

Expected Reproduction Time (in Minutes)

Setup: 5–10 min. Execution: < 5 min.

Artifact Setup (incl. Inputs)

Hardware. Any Linux system with Python environment.

Software.

- Python 3.8+
- matplotlib, pandas

Datasets / Inputs. Processed CSV files included.

Installation and Deployment. Install dependencies via `pip install -r requirements.txt`.

Artifact Execution

Workflow: T_1: Load CSVs. T_2: Generate figures and tables using Python scripts.

Artifact Analysis (incl. Outputs)

Plots and tables identical to those in the paper (Figs. 2–6, Table V).