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

Hammad Ather*, Chen Wang†, Hariharan Devarajan††, Hank Childs*, Kathryn Mohror††

**University of Oregon, †Nanyang Technological University, ††Lawrence Livermore National Laboratory*

PDSW 2025, St. Louis, MO

UNIVERSITY OF OREGON

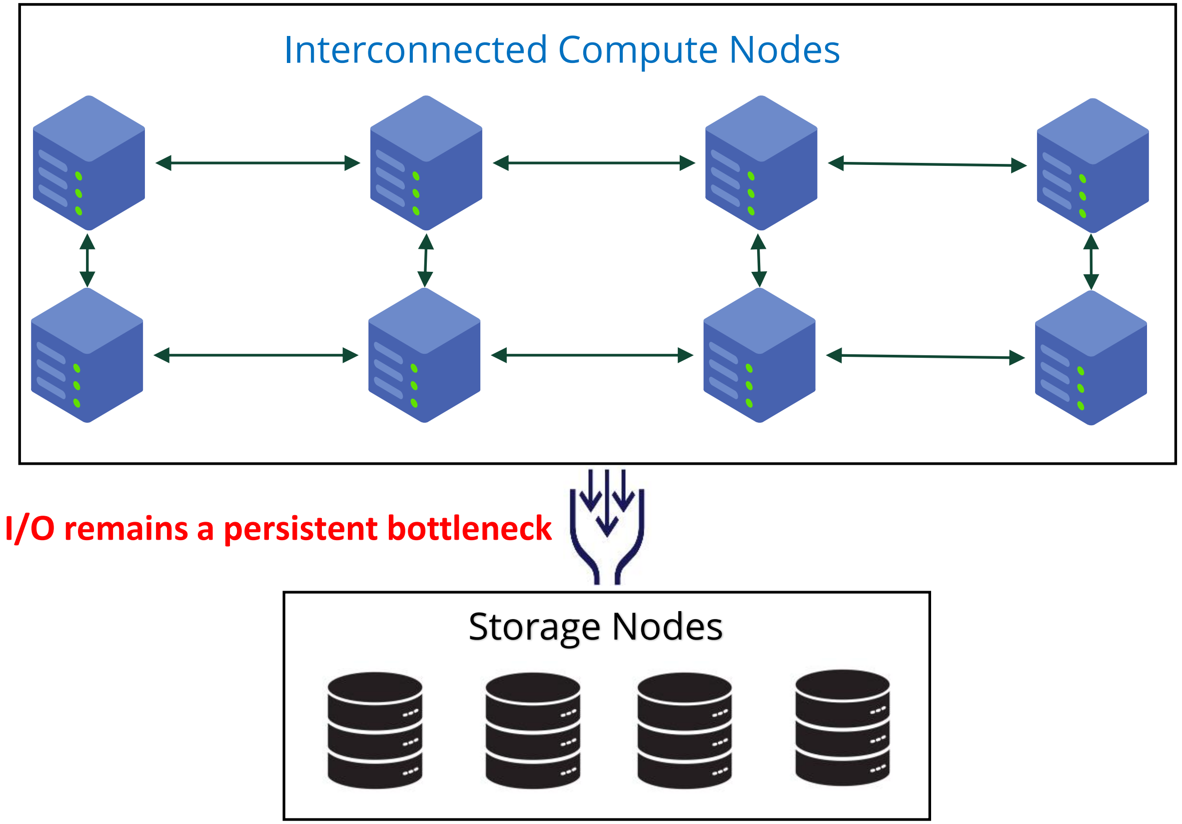
LLNL-CFPRES-2013617

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

- **Motivation and Background**
- *SmartIO:* An End-to-End Approach for Runtime I/O Optimization
- *Case Study 1:* Throughput Analysis with IOR Benchmark
- *Case Study 2:* Throughput Analysis with Flash-X
- Overhead Analysis
- Comparison with State-of-the-Art
- Conclusion and Future Work

Motivation

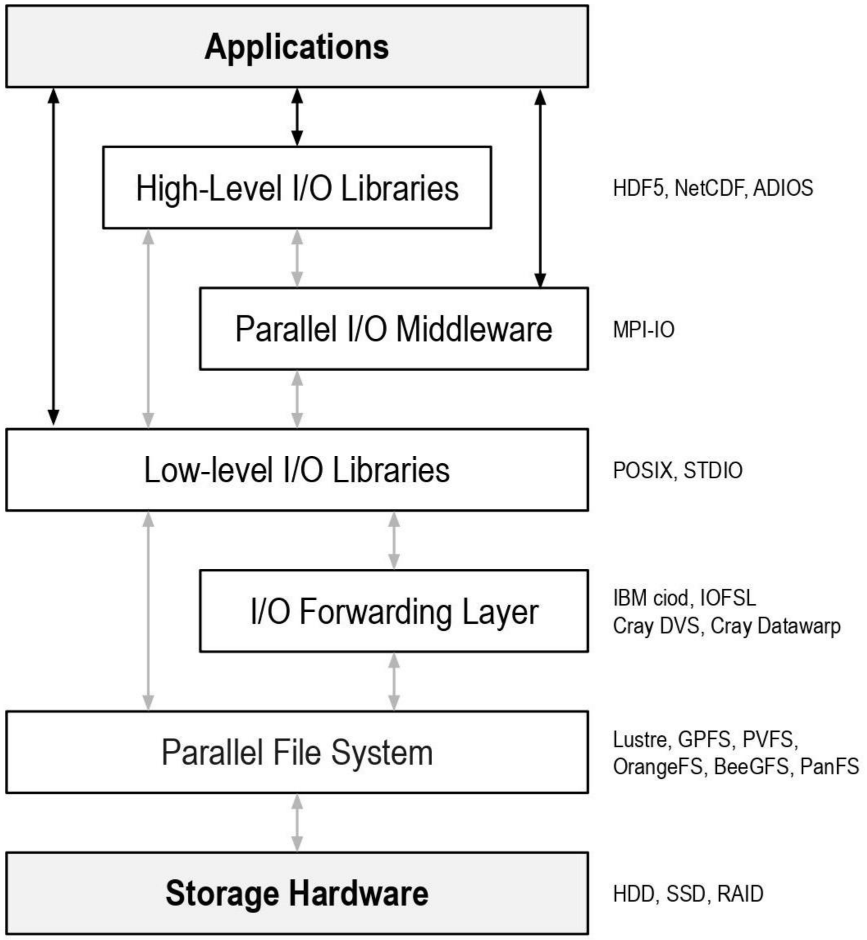
- HPC systems run **critical** workloads
  - Made up of interconnected nodes
- Compute power has advanced exponentially
  - The **I/O subsystem has not scaled** at the same pace
  - **Inefficient I/O** leads to idle compute resources and reduced overall performance



Why is I/O a Bottleneck?

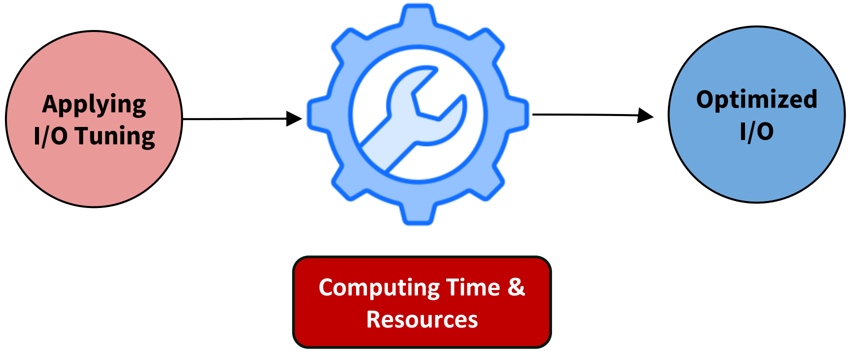
- HPC I/O stack is complex
  - I/O bottlenecks arise because data must move through the multiple layers
  - Bottleneck in any layer can propagate through the entire I/O path

How can I/O be optimized?



Optimizing Parallel I/O

- Significant efforts have been made to optimize I/O performance
  - **Offline Tuning**
    - Provide actionable recommendations; optimizations applied in future runs
  - **Autotuning**
    - Searches for the optimal parameter configuration; can require hours to converge
  - **ML-Based Optimization**
    - Relies on extensive training of ML models; data collected from multiple simulation runs
  - **Suffer from high resource and time overhead**

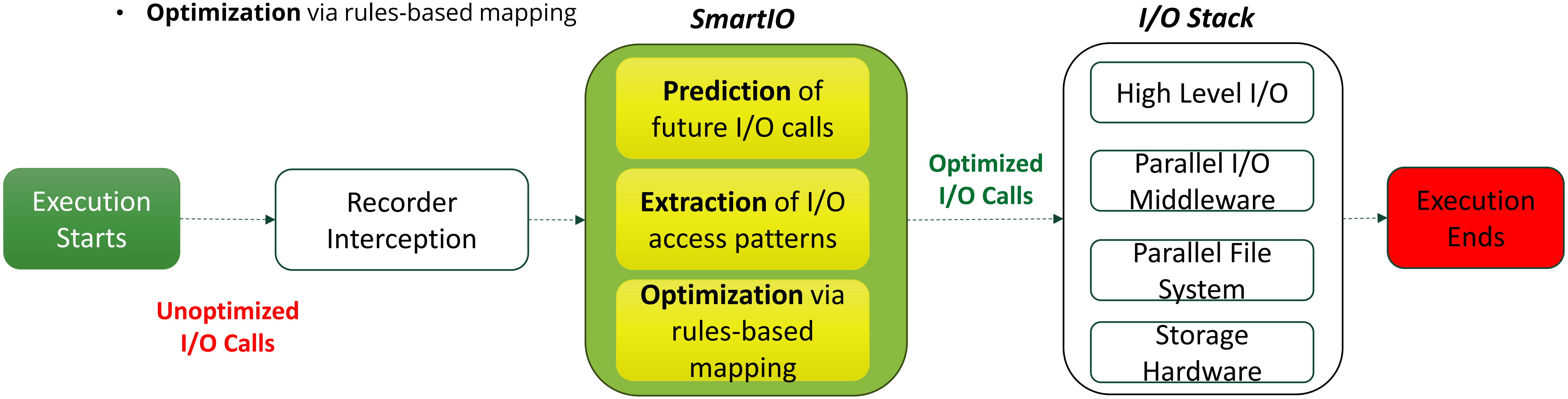


Can I/O be optimized with **minimal overhead**?

- Motivation and Background
- ***SmartIO:* An End-to-End Approach for Runtime I/O Optimization**
- *Case Study 1:* Throughput Analysis with IOR Benchmark
- *Case Study 2:* Throughput Analysis with Flash-X
- Overhead Analysis
- Comparison with State-of-the-Art
- Conclusion and Future Work

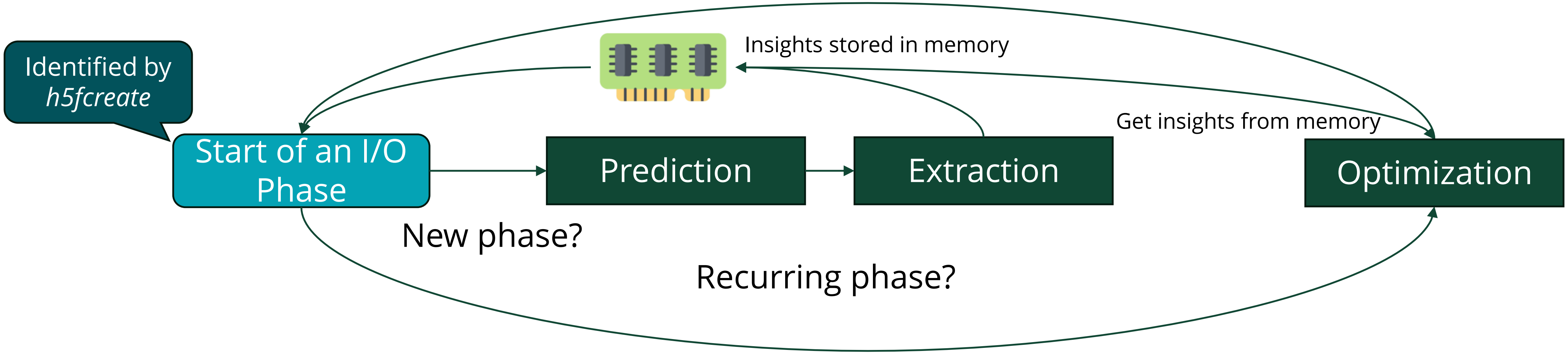
SmartIO: End-to-End Runtime I/O Optimization

- **Learns** how the application performs I/O, identifies **recurring** access patterns, and automatically **optimizes** them during execution – without requiring exhaustive parameter searches, training, or reruns
- Combines **three** components together into an end-to-end workflow:
  - **Prediction** of future I/O calls
  - **Extraction** of I/O access patterns
  - **Optimization** via rules-based mapping



Building the End-to-End Workflow

- HPC applications often execute **periodic I/O phases** with recurring patterns
- **Distinguishing** between new and recurring phases
    - Keeps track of all I/O files; string matching algorithm to determine new or previously seen file



First Component: Prediction

- **Pattern-Recognition**
  - Recorder uses context-free grammars (CFG) to identify **recurring** I/O patterns

Example I/O Code

```
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        read(fd, buf, size)
    }
    fsync(fd);
}
```

CFG

$$S \rightarrow A^m$$
$$A \rightarrow a^n\, b$$

- **I/O Call Sequence Prediction**
  - Prediction algorithm checks if a terminal symbol is the **start** of a recurring pattern
  - **Multiple** rules can be possible predictions
    - Weighing system to choose the most accurate prediction → Hash Table
    - Symbols mapped to function calls and stored in a stack

Second Component: Extraction

Table 1: List of all the function calls across different I/O libraries for extracting I/O insights from predicted calls

	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	✗

Second Component: Extraction

Table 1: List of all the function calls across different I/O libraries for extracting I/O insights from predicted calls

	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	✗

Second Component: Extraction

Table 1: List of all the function calls across different I/O libraries for extracting I/O insights from predicted calls

	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	✗

Third Component: Optimization

Table 2: The optimization rules for HDF5, ROMIO, and Lustre, including their sources from literature and empirical evaluations

Optimization	Rule	Evaluation
HDF5 Data Transfer Mode	**1.** Use Independent I/O for sequential reads/writes to a shared file [11, 18]	✗
	2. Use Collective I/O for random or non-sequential reads/writes to a shared file [11, 21, 30–32]	✗
	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 [32]	✗
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 with a shared file, otherwise keep default [18]	✓
ROMIO Data Sieving	**8.** Keep default values for the data sieving parameters [30]	✓
Lustre Stripe Count	**9.** For file-per-process configurations, set stripe count to 1 [1, 17, 22, 24, 34]	✗
	10. For a shared file, set a progressive stripe layout if Lustre version > 2.10; otherwise, set stripe count according to file size [1, 17, 22, 24, 26, 27, 34]	✓

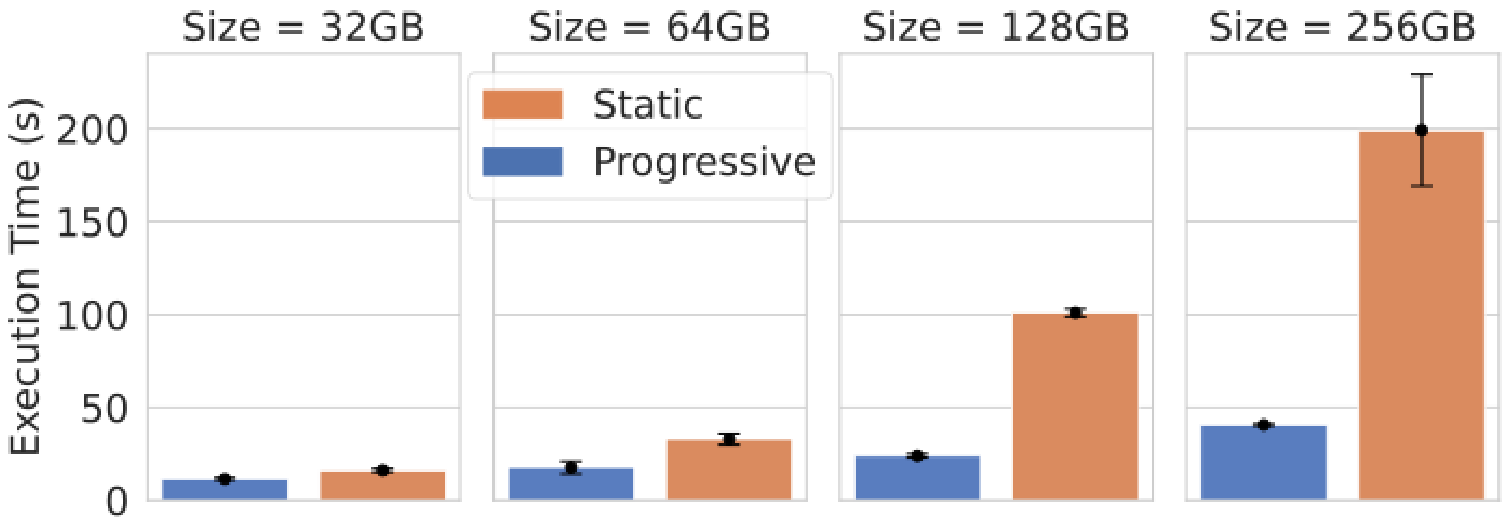
Third Component: Optimization

Table 2: The optimization rules for HDF5, ROMIO, and Lustre, including their sources from literature and empirical evaluations

Optimization	Rule	Evaluation
HDF5 Data Transfer Mode	**1.** Use Independent I/O for sequential reads/writes to a shared file [11, 18]	✗
	2. Use Collective I/O for random or non-sequential reads/writes to a shared file [11, 21, 30–32]	✗
	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 [32]	✗
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 with a shared file, otherwise keep default [18]	✓
ROMIO Data Sieving	**8.** Keep default values for the data sieving parameters [30]	✓
Lustre Stripe Count	**9.** For file-per-process configurations, set stripe count to 1 [1, 17, 22, 24, 34]	✗
	10. For a shared file, set a progressive stripe layout if Lustre version > 2.10; otherwise, set stripe count according to file size [1, 17, 22, 24, 26, 27, 34]	✓

Empirical Evaluations

- **Evaluations for Lustre Stripe Count**
  - **Significant** literature to guide in formulating the optimization rule for correct stripe count
  - However, should the striping be done **statically** or **progressively**?
  - Performed a **scaling** study with IOR, increasing the number of block sizes
  - Results showed that progressive striping performs better with **increasing** file sizes



- Motivation and Background
- *SmartIO:* An End-to-End Approach for Runtime I/O Optimization
- ***Case Study 1:* Throughput Analysis with IOR Benchmark**
- *Case Study 2:* Throughput Analysis with Flash-X
- Overhead Analysis
- Comparison with State-of-the-Art
- Conclusion and Future Work

Optimizing IOR at Runtime

- Performed a **scaling study** on Lassen and Ruby
  - No. of nodes: 8 to 32
  - MPI processes: 256 to 1024

Option	Value
API	HDF5
Access	Single-shared-file
Type	Collective
Ordering in File	Sequential
Request Type	Read and Write
Segments	1
Block Size	128M
Transfer Size	4M
Timesteps	7

IOR configuration for the scaling study

Mapping Insights to Optimizations

Predicted Insights

Collective I/O

Shared File I/O

Sequential Accesses

Transfer Size < 4MB

Lustre PFS

Applied Optimizations

Changed **data transfer mode** from collective to independent

Mapping Insights to Optimizations

Predicted Insights

Collective I/O

Shared File I/O

Sequential Accesses

Transfer Size < 4MB

Lustre PFS

Applied Optimizations

Changed **data transfer mode** from collective to independent

Switched **romio_cb_write** from automatic to disable

Mapping Insights to Optimizations

Predicted Insights

- Collective I/O
- Shared File I/O
- Sequential Accesses
- Transfer Size < 4MB
- Lustre PFS

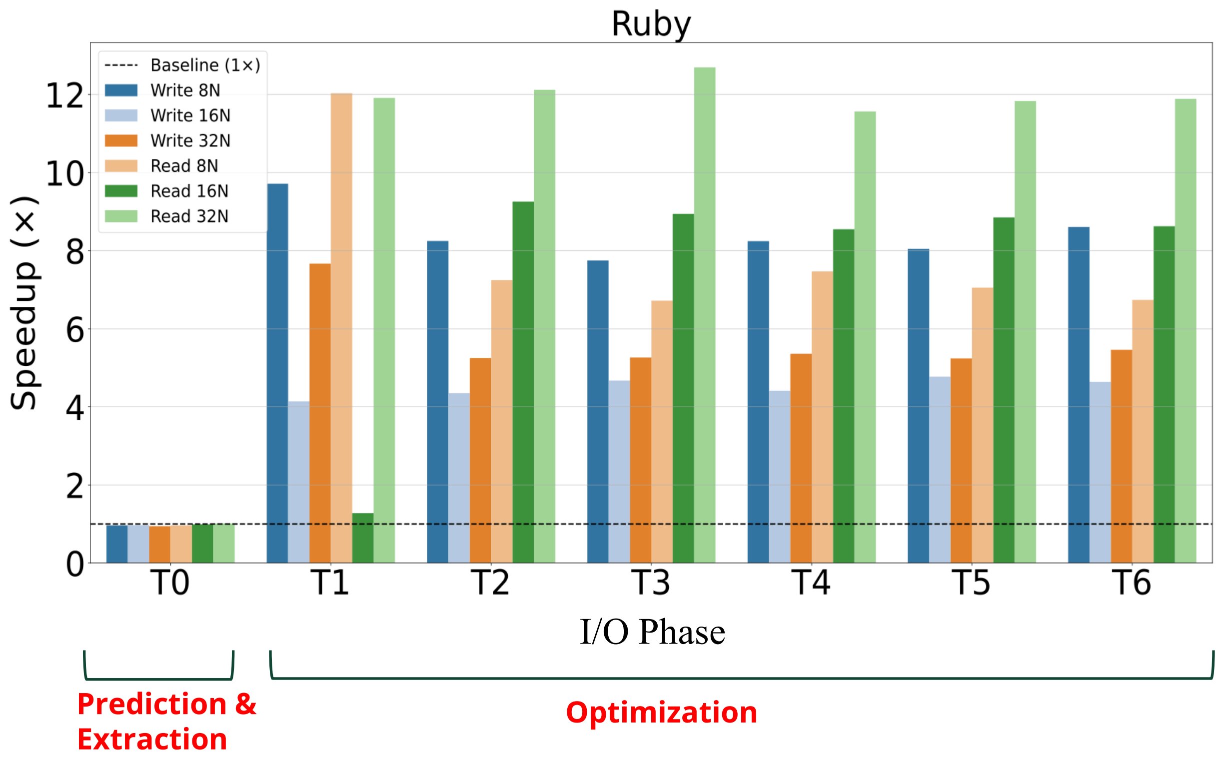
Applied Optimizations

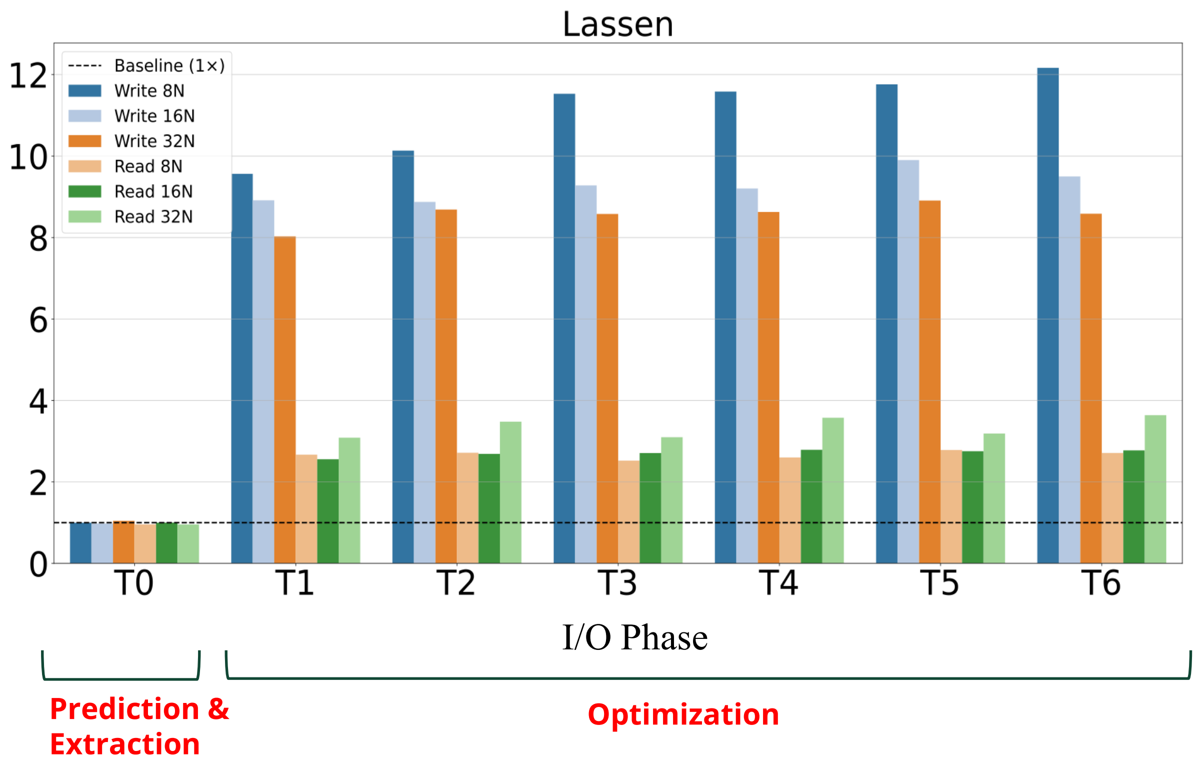
- Changed **data transfer mode** from collective to independent
- Switched **romio_cb_write** from automatic to disable
- Changed **metadata cache** size from 2MB to 8MB
- Changed from no **alignment** to 1MB

Mapping Insights to Optimizations

Predicted Insights	Applied Optimizations
Collective I/O	Changed **data transfer mode** from collective to independent
Shared File I/O	Switched **romio_cb_write** from automatic to disable
Sequential Accesses	Changed **metadata cache** size from 2MB to 8MB
Transfer Size < 4MB	Changed from no **alignment** to 1MB
Lustre PFS	Set **progressive** striping

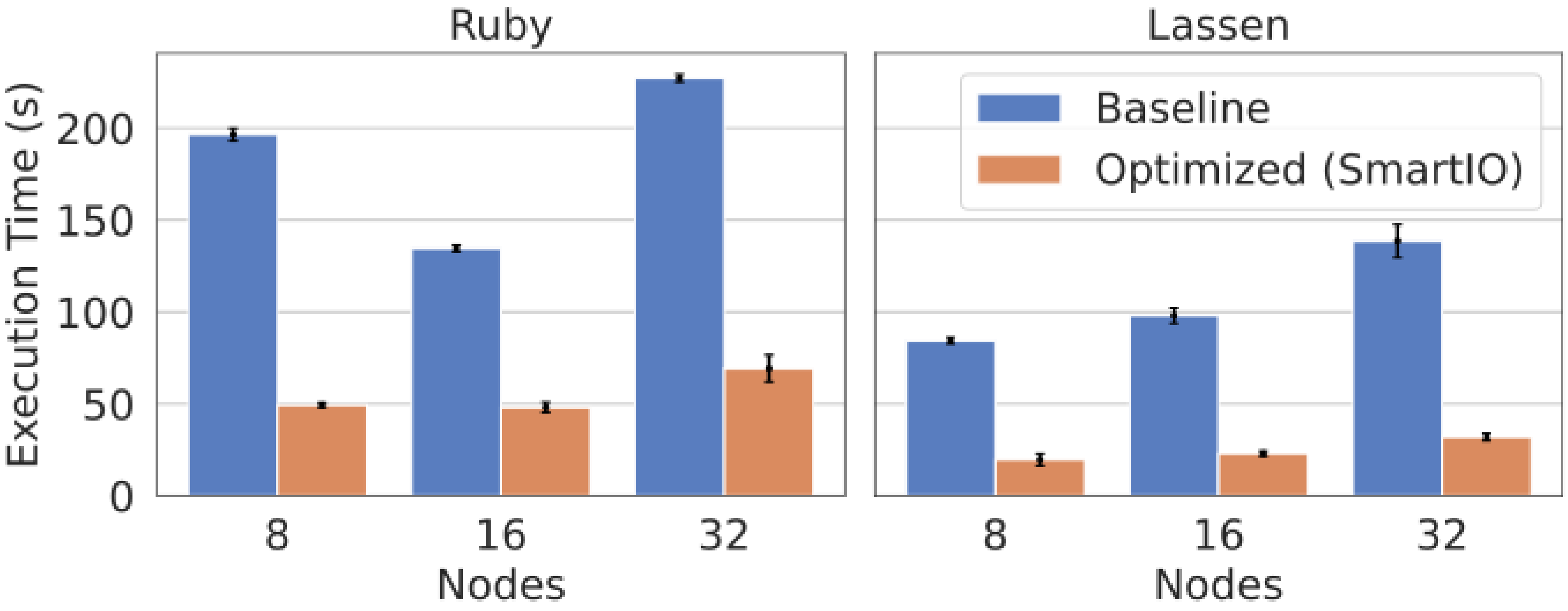
IOR Bandwidth Speedup





- 13x read & 12x write bandwidth speedup
  - < 1s of tuning overhead

IOR Execution Times



- Motivation and Background
- *SmartIO:* An End-to-End Approach for Runtime I/O Optimization
- *Case Study 1:* Throughput Analysis with IOR Benchmark
- ***Case Study 2:* Throughput Analysis with Flash-X**
- Overhead Analysis
- Comparison with State-of-the-Art
- Conclusion and Future Work

Optimizing Flash-X at Runtime

- Performed a **scaling study** on Ruby
  - No. of nodes: 8 to 32
  - MPI processes: 448 to 1792

Parameter	Purpose	Value
nblockx	*x* dimension blocks	8,16,32
nblocky	*y* dimension blocks	8,16,32
nblockz	*z* dimension blocks	8,16,32
checkpointFileIntervalStep	Simulation steps between checkpoints	10
nend	Total simulation steps	60
tmax	Maximum simulation time	500000
lrefine_max	Levels of adaptive mesh refinement (AMR)	6

Flash-X configuration for the scaling study

Mapping Insights to Optimizations

Predicted Insights

- Collective & Independent I/O → Updated **cb_config_list** from 1 to 8
- Transfer Size <= 1MB
- Shared File I/O
- Lustre PFS

Applied Optimizations

- Updated **cb_config_list** from 1 to 8

Mapping Insights to Optimizations

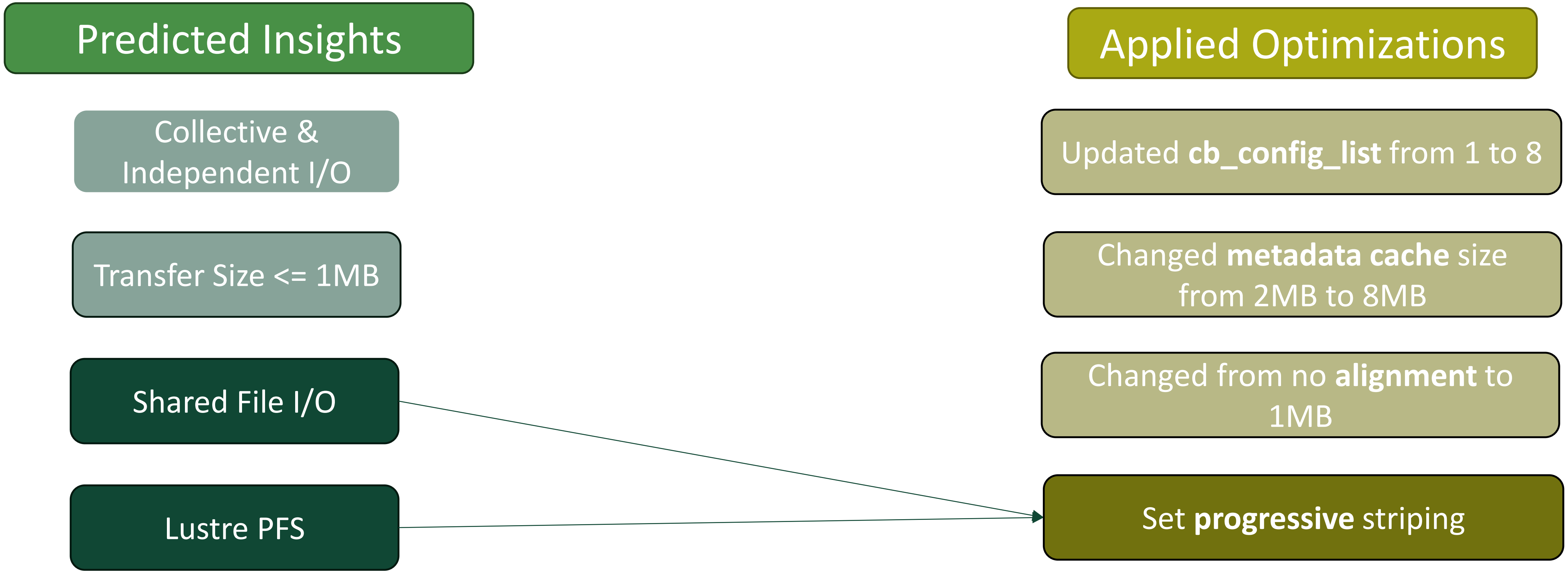
Predicted Insights

- Collective & Independent I/O
- Transfer Size <= 1MB
- Shared File I/O
- Lustre PFS

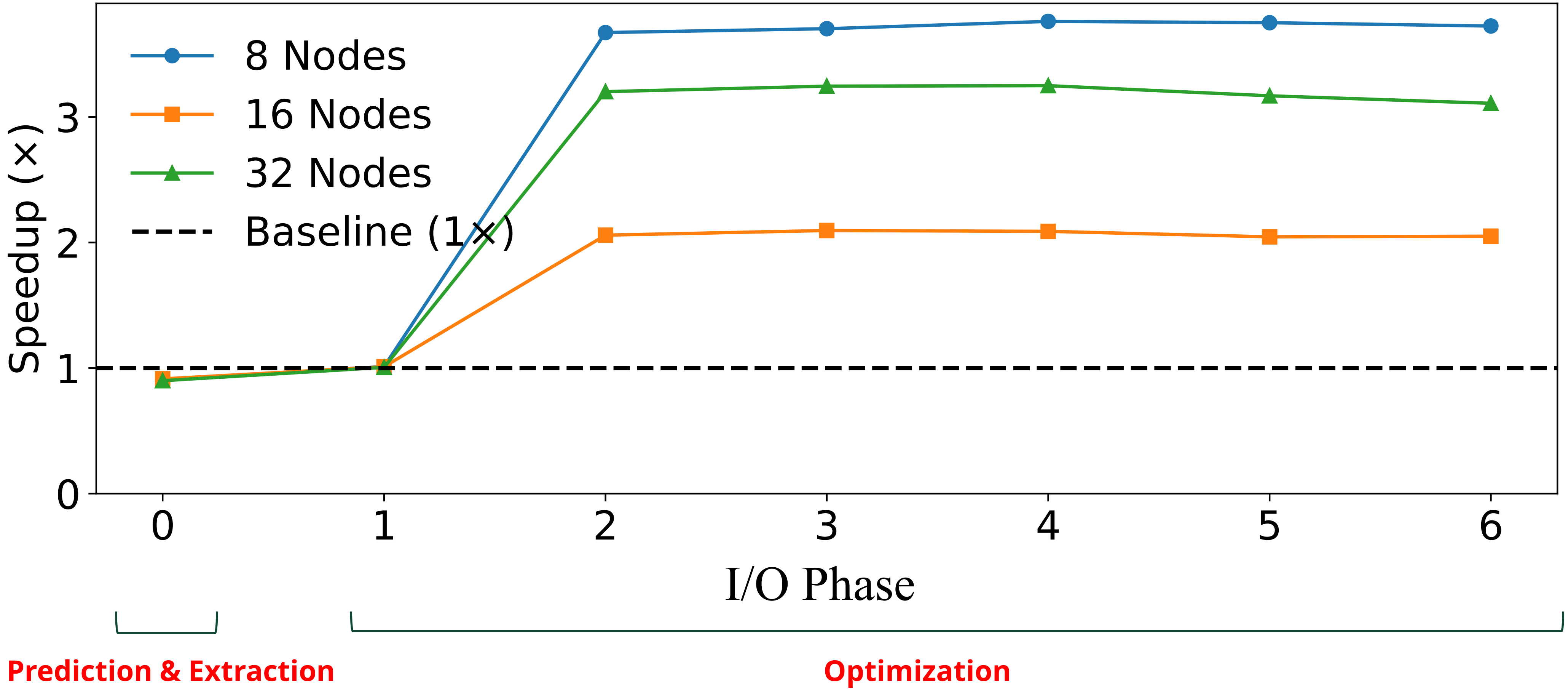
Applied Optimizations

- Updated **cb_config_list** from 1 to 8
- Changed **metadata cache** size from 2MB to 8MB
- Changed from no **alignment** to 1MB

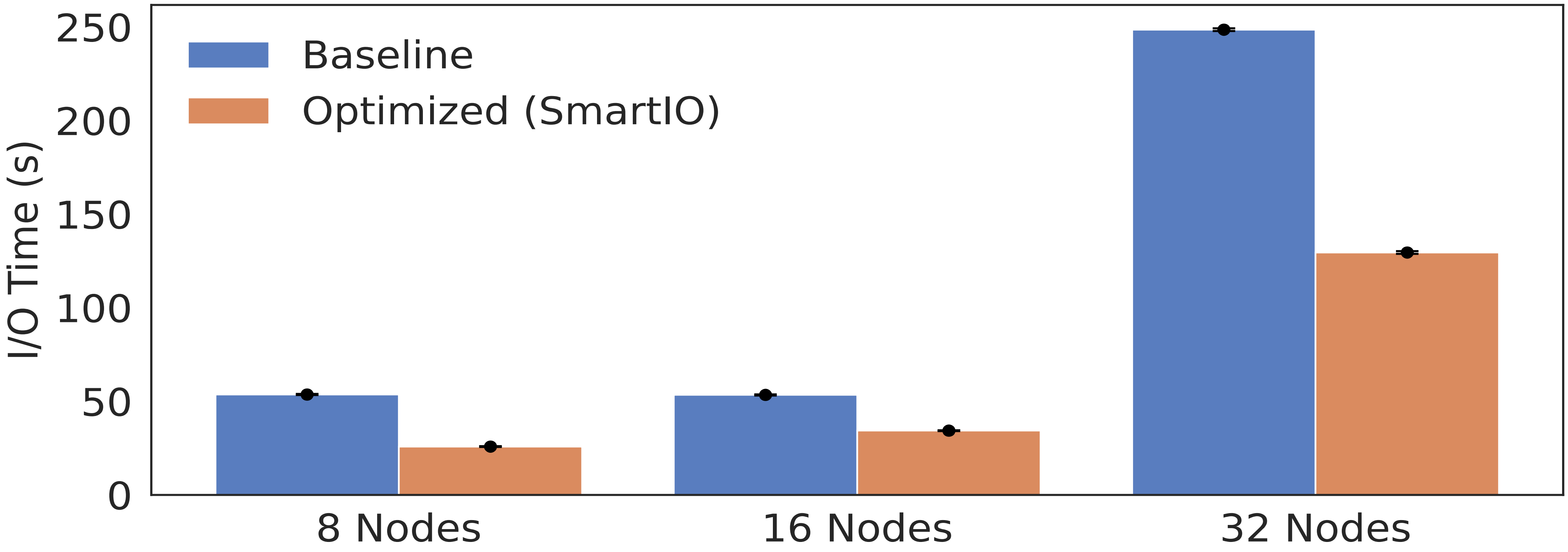
Mapping Insights to Optimizations



Flash-X Bandwidth Speedup



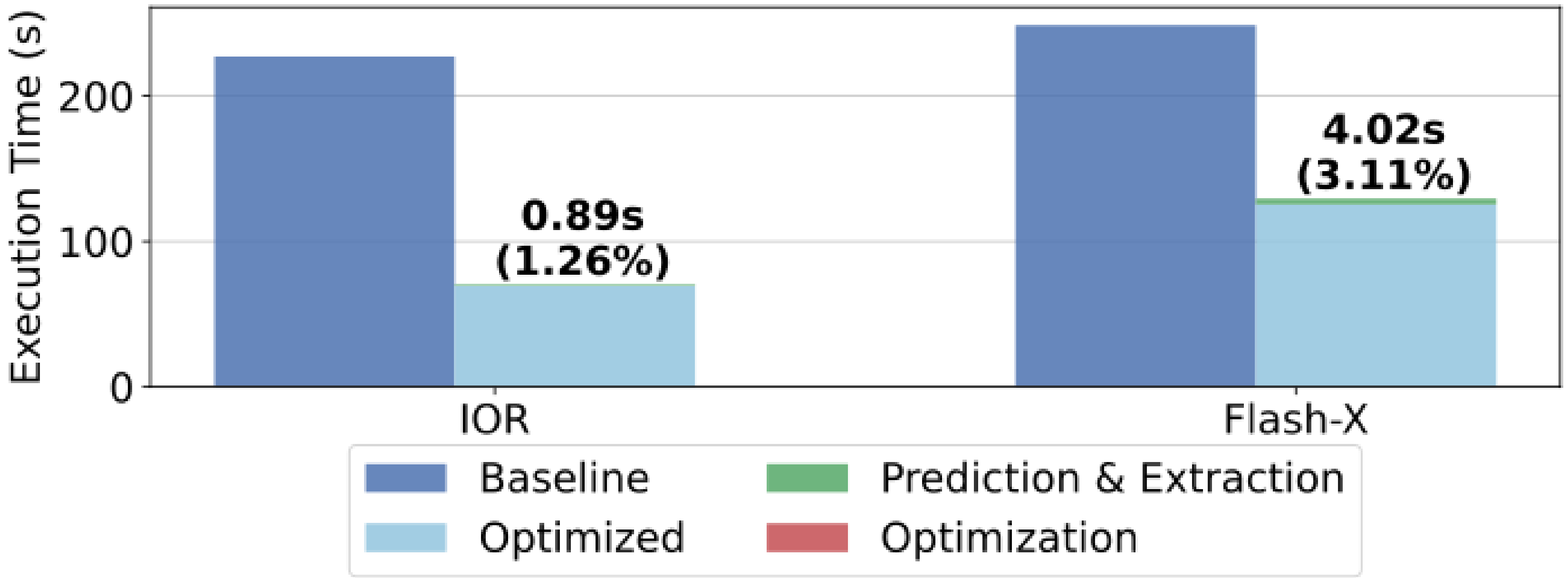
Flash-X I/O Times



- ~50% I/O time reduction
  - < 4s of tuning overhead

- Motivation and Background
- *SmartIO:* An End-to-End Approach for Runtime I/O Optimization
- *Case Study 1:* Throughput Analysis with IOR Benchmark
- *Case Study 2:* Throughput Analysis with Flash-X
- **Overhead Analysis**
- Comparison with State-of-the-Art
- Conclusion and Future Work

Overhead



Execution Time (s)
0
100
200
0.89s
(1.26%)
4.02s
(3.11%)
IOR
Flash-X
Baseline
Optimized
Prediction & Extraction
Optimization

- Motivation and Background
- *SmartIO:* An End-to-End Approach for Runtime I/O Optimization
- *Case Study 1:* Throughput Analysis with IOR Benchmark
- *Case Study 2:* Throughput Analysis with Flash-X
- Overhead Analysis
- **Comparison with State-of-the-Art**
- Conclusion and Future Work

Comparison with State-of-the-Art

- Compared **speedup** and **tuning overhead** with autotuning frameworks
- Ran *SmartIO* on **same** IOR configuration

Tuning Tool	Speedup (×)	Tuning Overhead (s)
Autotuning Framework [5, 6]	9	36000
SmartIO	6	0.08

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

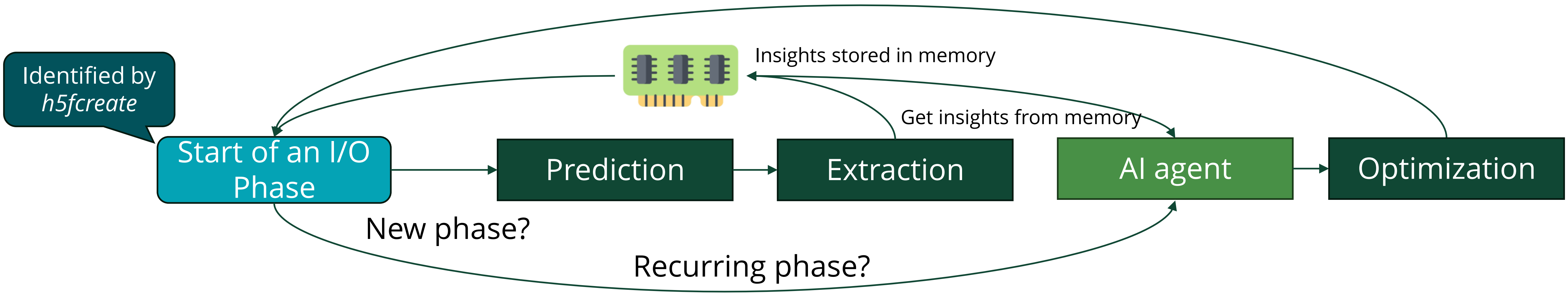
- Motivation and Background
- *SmartIO:* An End-to-End Approach for Runtime I/O Optimization
- *Case Study 1:* Throughput Analysis with IOR Benchmark
- *Case Study 2:* Throughput Analysis with Flash-X
- Overhead Analysis
- Comparison with State-of-the-Art
- **Conclusion and Future Work**

Conclusion

- **Challenge:** State-of-the-art optimization approaches are time and resource intensive
- This paper introduces ***SmartIO***, a lightweight end-to-end workflow for runtime I/O optimization
    - Can optimize the different layers of the I/O stack **without** prior training, profiling, or parameter searches

Future Work

- How can AI/LLMs assist in parallel I/O optimization?



Thank you!

UNIVERSITY OF
OREGON