Enhancing I/O performance: Leveraging Runtime and Offline Optimization Frameworks

Hammad Ather[1], Jean Luca Bez[2], Chen Wang[3], Hariharan Devarajan[3], Suren Byna[2], Kathryn Mohror[3], Hank Childs[1], Allen D. Malony[1]

University of Oregon[1], Lawrence Berkeley National Laboratory[2], Lawrence Livermore National Laboratory[3]

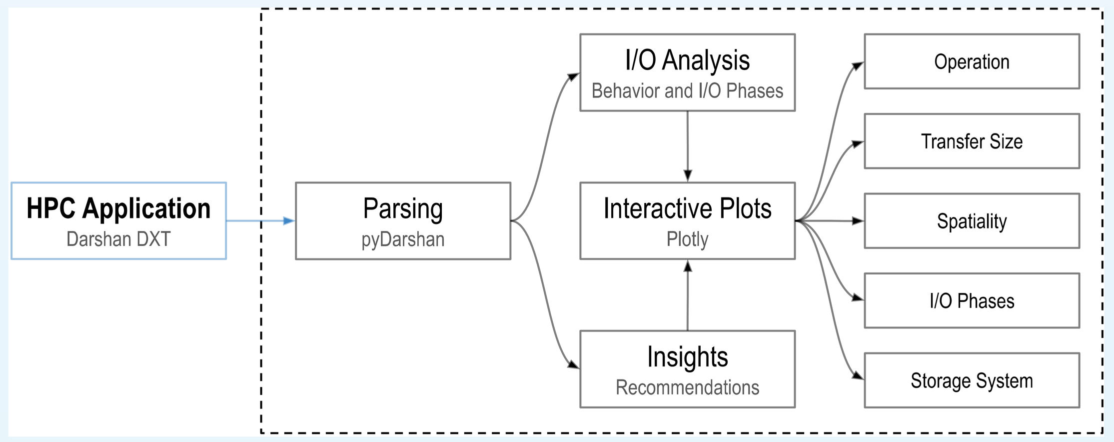
ABSTRACT

In this dissertation we propose two different frameworks, one for offline and one for online I/O optimization. The offline framework, called *Drishti IO*, offers interactive visualizations of an application's I/O behavior. It identifies root causes of I/O bottlenecks and provides actionable recommendations to enhance performance. The online framework builds upon the Recorder I/O tracing tool, introducing a runtime I/O prediction and optimization system. This framework utilizes context-free grammars to optimize I/O behavior dynamically during runtime. With this detailed analysis and real-time optimizations, these frameworks present comprehensive approach to improving I/O performance.

OPTIMIZING I/O OFFLINE

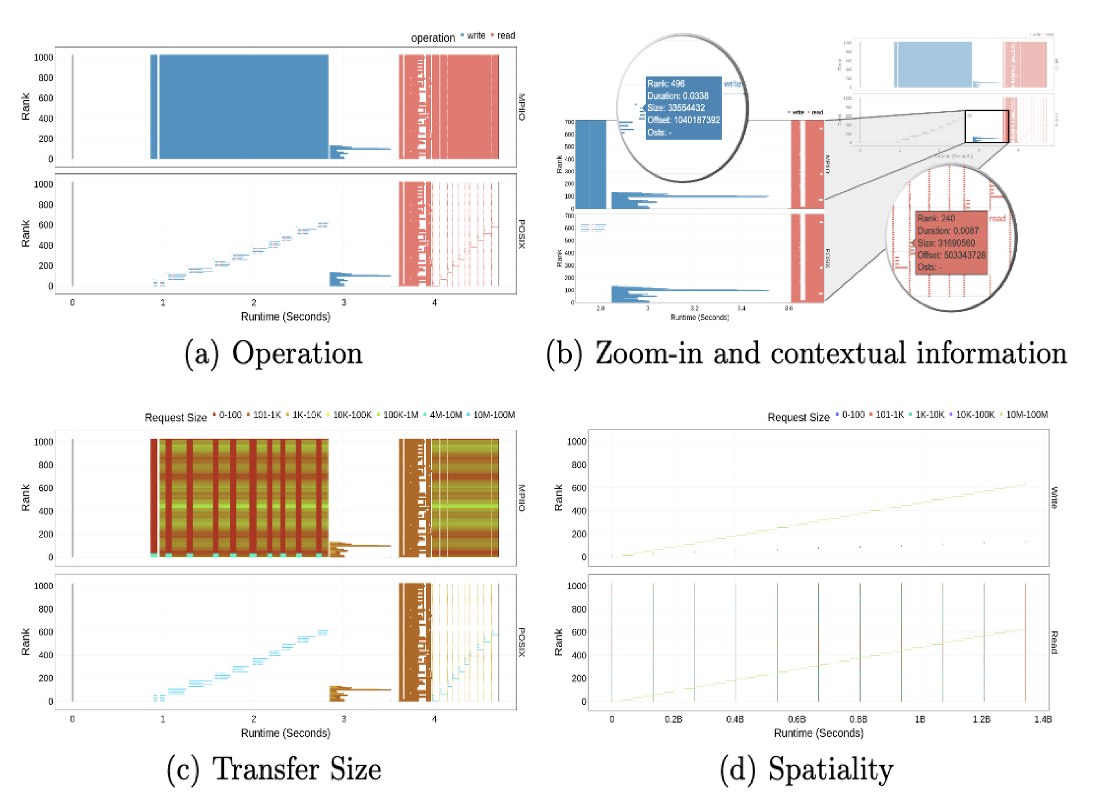
The offline framework (*Drishti IO*) has two key features:

- **Interactive visualizations**
- **Automatic detection of I/O bottlenecks**

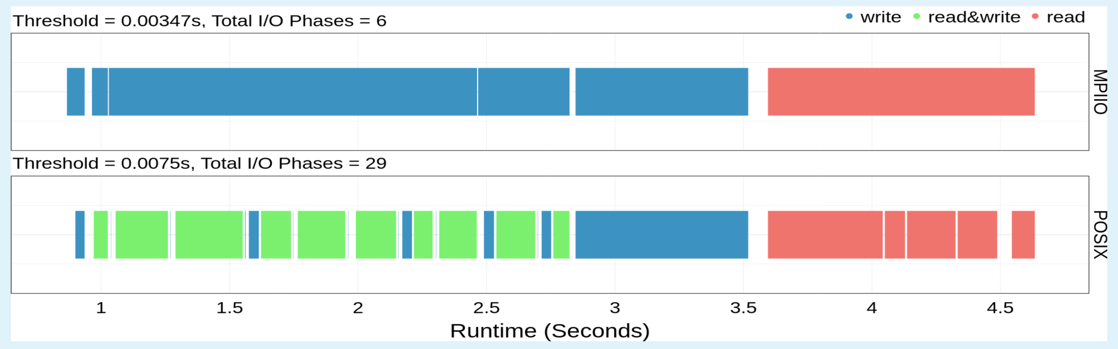


Drishti generates meaningful interactive visualizations and a set of recommendations based on the detected I/O bottlenecks using Darshan DXT I/O trace

INTERACTIVE VISUALIZATIONS



Interactive visualizations 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..



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

AUTOMATIC DETECTION OF I/O BOTTLENECKS

Drishti IO can also:

- **Diagnose I/O bottlenecks**
- **Provide actionable recommendations**
- **Drill down to the source code**

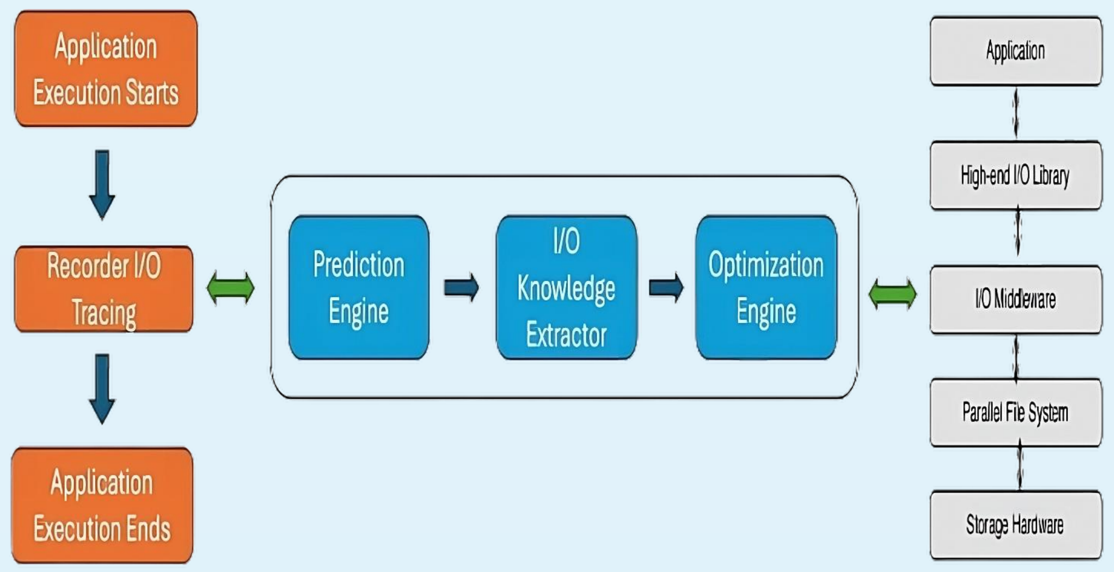
```
RECORDER | 2 critical issues | 2 warnings | 8 recommendations

▸ 260 files (0 use STDIO, 1 use POSIX, 10 use MPI-IO)
▸ Application is write operation intensive (99.87% writes vs. 0.13% reads)
▸ Application is write size intensive (100.00% write vs. 0.00% read)

▸ High number (491640) of small read requests (< 1MB)
    ▸ 99.99% of all read requests
    ▸ Observed in 10 files:
        ▸ plt0000[0-9].h5 with 49164 (10.00%) small write requests
    ▸ Recommended action:
        ▸ Consider buffering read operations into larger, contiguous ones
        ▸ Since the application uses MPI-IO, consider using collective I/O calls
          to aggregate requests into larger, contiguous ones
          (e.g. MPI_File_write_all() or MPI_File_write_at_all())
▸ Detected data transfer imbalance caused by stragglers
    ▸ Observed in 10 shared files:
        ▸ plt0000[0-9].h5 with 100.00% imbalance
    ▸ Recommended action:
        ▸ Consider better balancing the data transfer between the application ranks
        ▸ Consider tuning the file system stripe size and stripe count

▸ Application could benefit from non-blocking (asynchronous) reads
    ▸ Recommended action:
        ▸ Since the application uses HDF5, consider using the ASYNC I/O VOL connector
        ▸ Since the application uses MPI-IO, consider non-blocking I/O operations
▸ Application could benefit from non-blocking (asynchronous) writes
    ▸ Recommended action:
        ▸ Since the application uses HDF5, consider using the ASYNC I/O VOL connector
        ▸ Since the application uses MPI-IO, consider non-blocking I/O operations

▸ Application mostly uses consecutive (3.16%) and sequential (6.33%) read requests
▸ Application mostly uses consecutive (99.98%) and sequential (0.01%) write requests
▸ Application uses MPI-IO and write data using 15360 (99.81%) collective operations
```

Drishti IO report and insights generated for AMReX in Perlmutter (NERSC) based on Recorder metrics/traces

THE RUNTIME I/O PREDICTION AND OPTIMIZATION ENGINE

The runtime I/O optimization framework is based on these modules:

- **Prediction Engine:** Predicts future I/O function calls using the context free grammar
- **I/O Knowledge Extractor:** Extracts I/O access patterns from the predicted future calls
- **Optimization Engine:** Maps I/O knowledge to optimizations for the predicted future calls

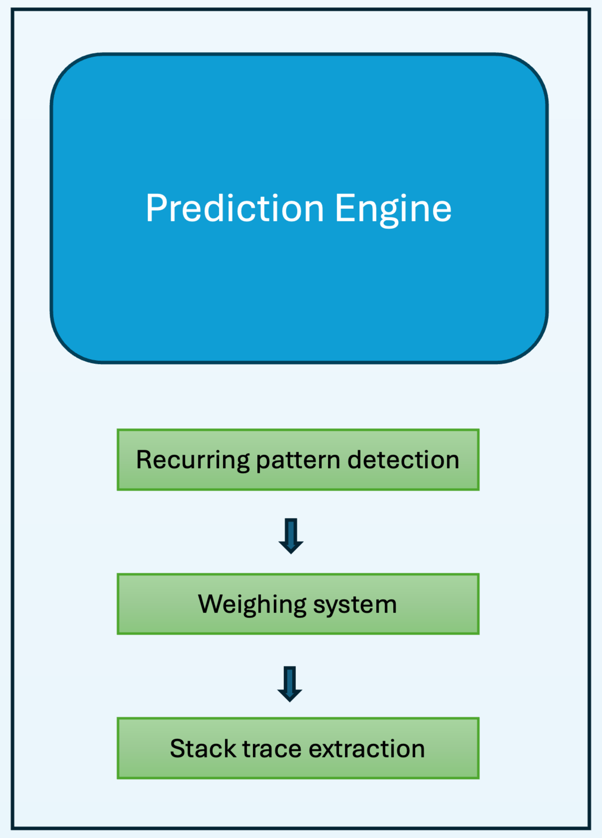


The proposed runtime I/O prediction and optimization engine. The engine comprises of three modules: (1) Prediction Engine, (2) I/O Knowledge Extractor, and (3) Optimization Engine

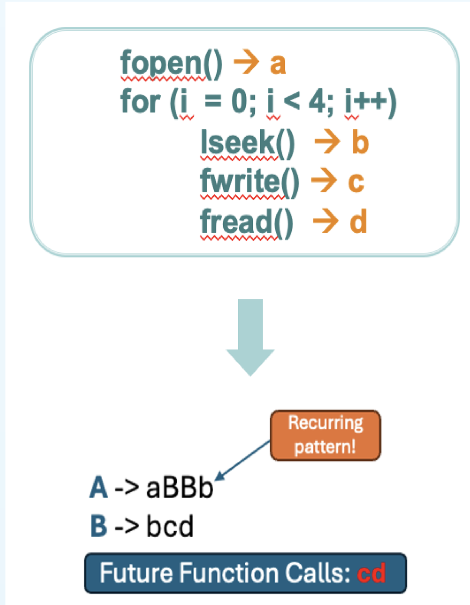
PREDICTION ENGINE

The prediction engine uses the context free grammar generated by Recorder to:

- Detect a recurring pattern
- Maintain a weighing system to predict the future function calls
- Maintain a stack trace of these calls for analysis



The proposed prediction engine and some of its key components

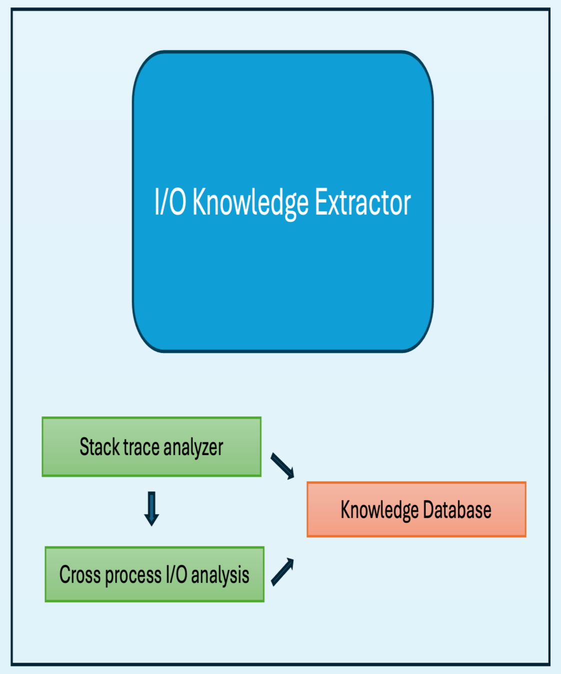


An example of how the prediction engine predicts future I/O calls from the CFG

"I/O KNOWLEDGE" EXTRACTOR

The I/O knowledge extractor performs the following tasks:

- Analyze the stack trace of predicted function calls
- Extract I/O access patterns per process and across processes
- Maintain a knowledge "database" for mapping these patterns to optimizations



The proposed "I/O Knowledge" extractor and some of its key components

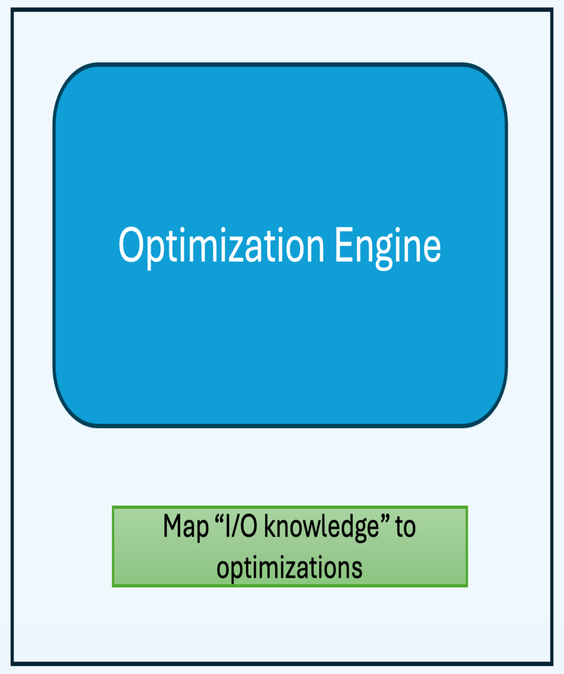
```
struct Knowledge {
    collective;
    operation;
    spatial_locality;
    file_name;
    file_operation;
};
```

Structure used to store per process "knowledge"

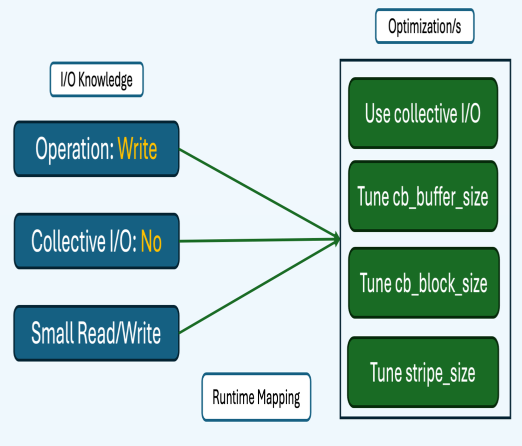
OPTIMIZATION ENGINE

The optimization engine is responsible for:

- Mapping the "I/O knowledge" collected by the extractor engine to optimizations
- Applying those optimizations at runtime

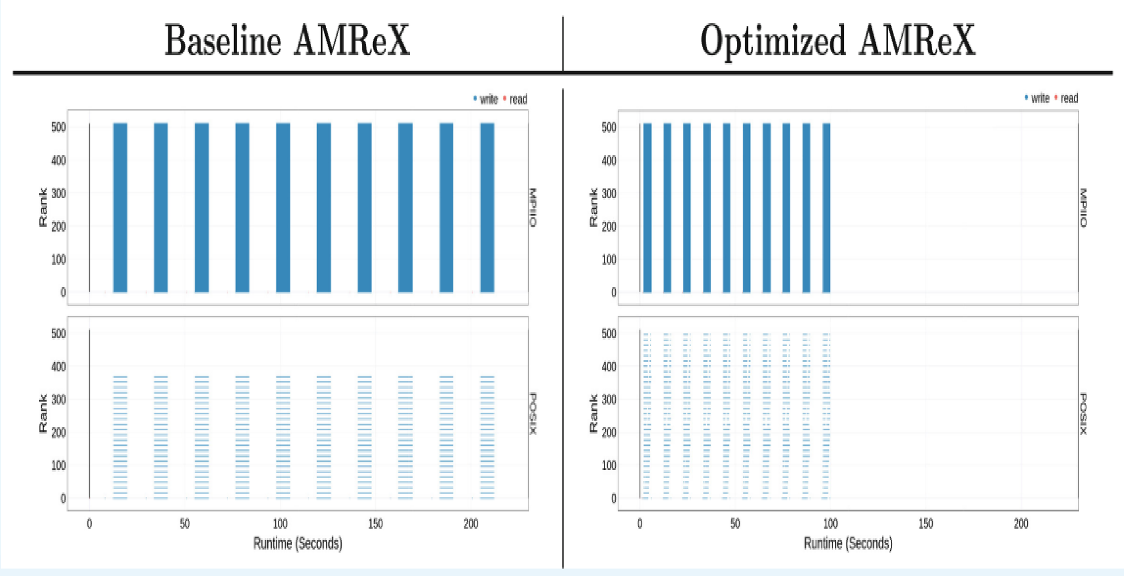


The proposed optimization engine and some of its key components

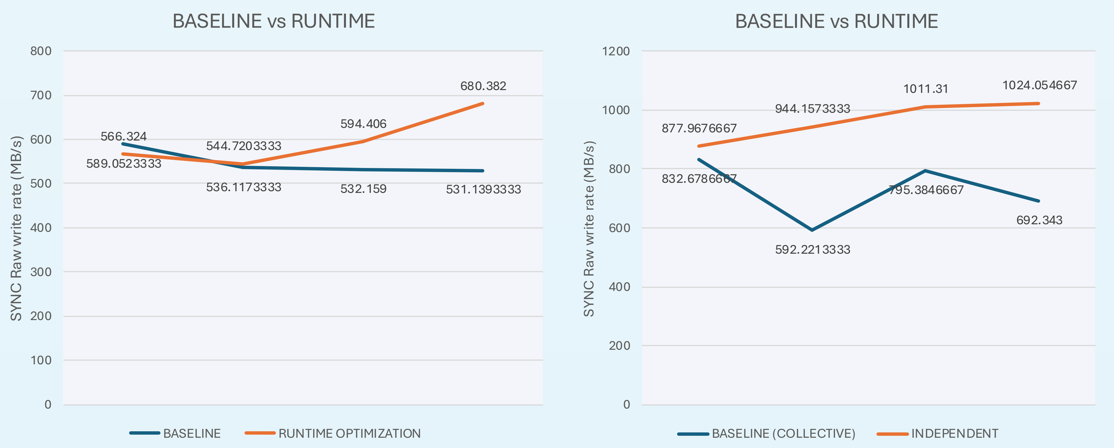


An example of how "I/O Knowledge" is mapped to optimizations

EXPERIMENTS



Comparison between the baseline and the optimized version of AMReX after applying the recommendations provided by *Drishti*



Comparison between the baseline and the optimized version of h5bench write, showing the increase in write throughput at runtime after applying the optimizations

CONCLUSION

This dissertation presents two novel frameworks to optimize I/O performance at both runtime and offline. The analysis done by these frameworks shows significant improvement in I/O throughput at both runtime and in the future runs of the application by applying the optimizations suggested by our frameworks.

ACKNOWLEDGMENT

This research was supported in part by the Exascale Computing Project (17-SC-20-SC). It was also supported by LBNL under contract no DE-AC02-05CH11231. This work is currently being performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE- AC52-07NA27344 (LLNL-POST-).