Recorder 2.0: Efficient Parallel I/O Tracing and Analysis

Chen Wang, Jinghan Sun and Marc Snir

Department of Computer Science

University of Illinois at Urbana-Champaign

Kathryn Mohror and Elsa Gonsiorowski

Center for Applied Scientific Computing

Lawrence Livermore National Laboratory

Contact: Chen Wang (chenw5@Illinois.edu)
Code: https://github.com/uiuc-hpc/Recorder

Motivation

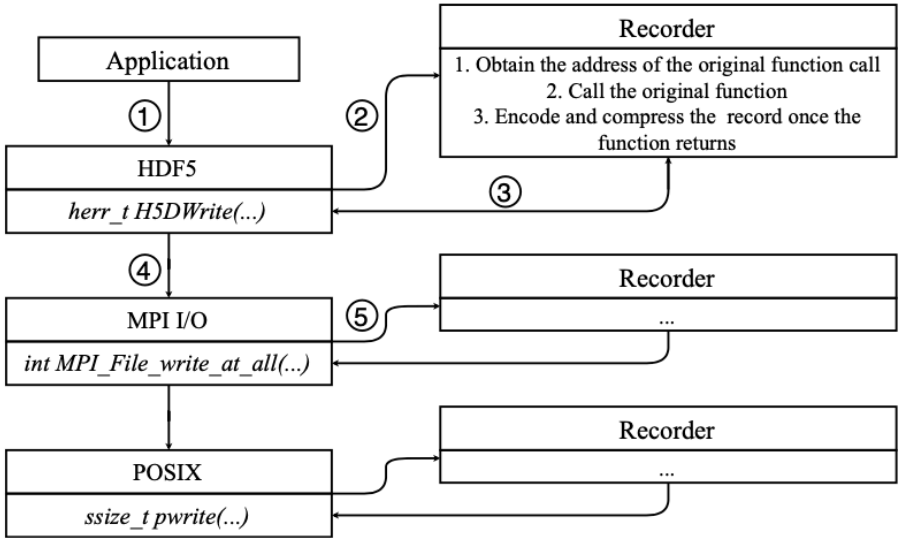
- Motivating questions:
  - What are the common access patterns of HPC applications?
  - Which functions and POSIX features do applications utilize?
  - To what extent can POSIX semantics be relaxed without affecting applications?
- Solution: Recorder collects all parameters to POSIX I/O operations so that file system developers can see the details of the I/O behaviors of applications.

Overview

- Recorder is a multi-level I/O tracing tool that captures HDF5, MPI-I/O, and POSIX I/O calls.
- Recorder 2.0 is a major update of the previous work in Recorder 1.0.
- Recorder faithfully keeps all parameters of every I/O function call.
- Recorder does not require modifications of application's code.
- Recorder uses a compact encoding schema and a on-the-fly decompression technique for post-processing.
- Recorder has a similar overhead in comparison with Score-p while keeping more details of I/O operations.

Instrumentation Framework

- Recorder is built as a shared library so that no code modifications or re-compilations are required.
- Need to be preloaded to intercept function calls.
- Functions intercepted by Recorder will be re-routed to the tracing process.
- Once the tracing process finished, Recorder will invoke the original function call.
- Recorder waits for the original function call to finish to update the exit timestamp.



Compact Tracing Format

- Recorder supports four tracing formats:
  - Plain text format
  - Binary format
  - Recorder format (compressed binary format)
  - zlib format (binary format + zlib compression)
- Recorder format:
  - Sliding window compression technique. Only keeps the differences from the referenced record.
  - status: indicate if the current record is compressed
  - Δtstart and Δtend: seconds elapsed from the starting timestamp.
  - ref_id: the reference record
  - diff_args: the different arguments that we need to store.

status	Δtstart	Δtend	ref_id	$diff\_arg_1$	...	$diff\_arg_n$
1Byte	4Bytes	4Bytes	1Byte	variable		

On-the-fly Decompression

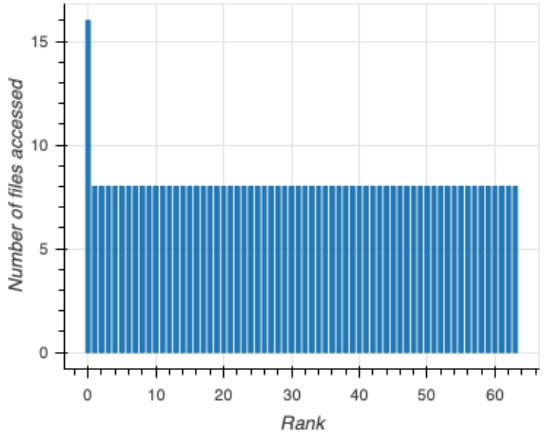
- *LOAD()* reads one field of an uncompressed record.
- Line 10: We only decompress a record if it is needed by the analysis.

Algorithm 1 Compute average read bandwidth

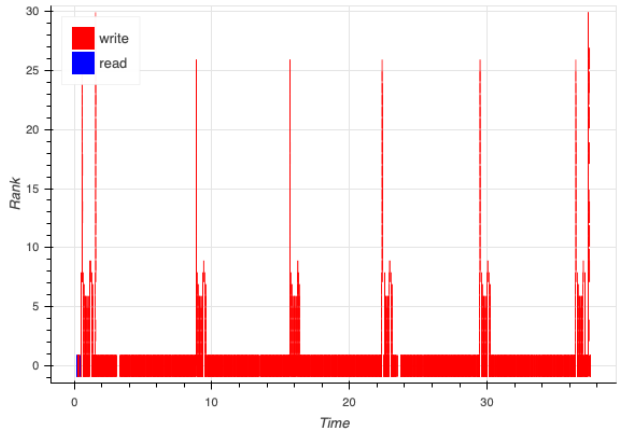
```
1:  for each rank do
2:      total_bytes = 0
3:      total_time = 0 + ϵ
4:      for each record do
5:          status ← LOAD(record, "status")
6:          if COMPRESSED(status) then
7:              func_id ← LOAD(ref_record, "func_id")
8:          else
9:              func_id ← LOAD(record, "func_id")
10:         if func_id in {pread, read, readv, etc} then
11:             if COMPRESSED(status) then
12:                 DECOMPRESS(record)
13:             Δtstart ← LOAD(record, "Δtstart")
14:             Δtend ← LOAD(record, "Δtend")
15:             total_bytes += LOAD(record, "bytes")
16:             total_time += (Δtstart − Δtend) ∗ T_R
17: avg_bandwidth ← total_bytes/total_time
```

Built-in Visualizations

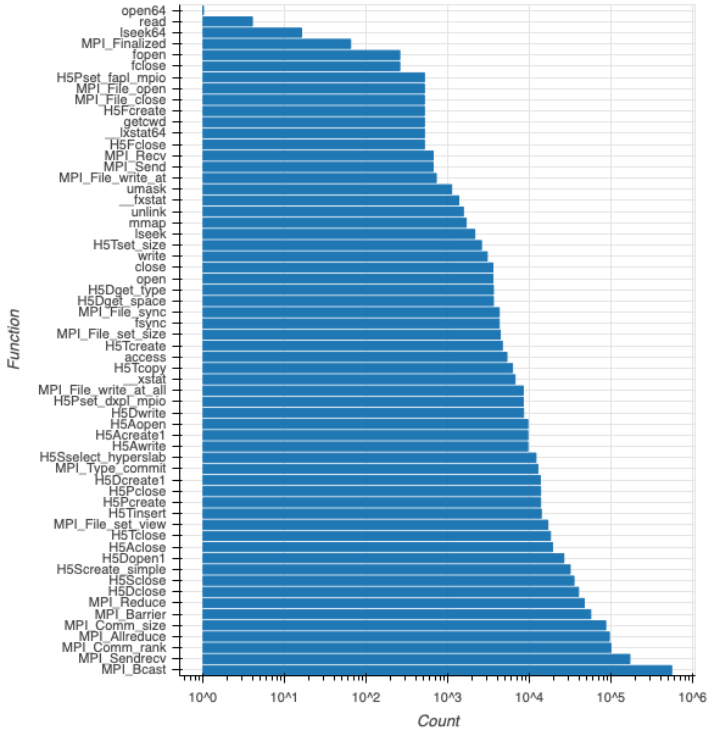
Example visualizations from the FLASH application:



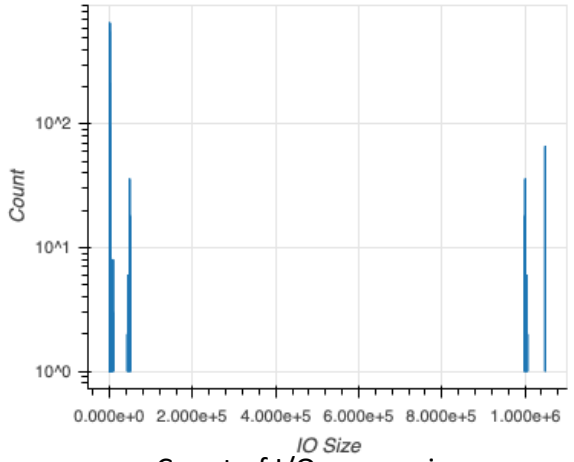
Number of files accessed by each rank



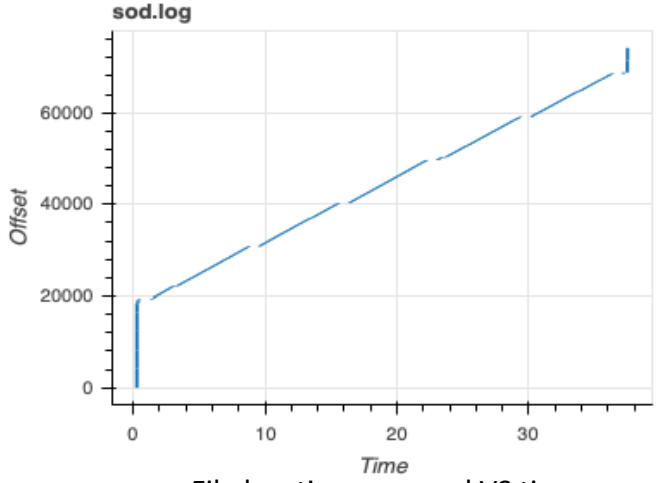
Overall I/O activity



Function Count



Count of I/O access sizes



File location accessed VS time

Evaluation

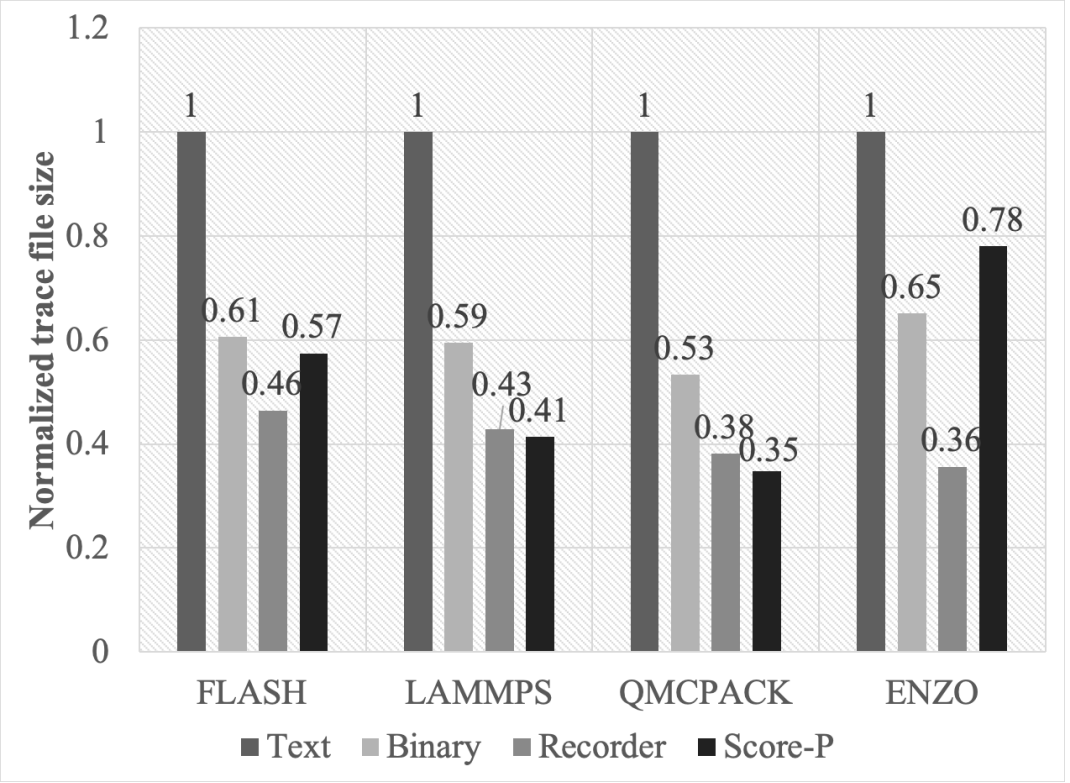
- Hardware:
  - Stampede2 at TACC
  - 24 SKX nodes with 24 ranks per node
  - Each node has 48 cores, 192GB DDR-4 memory, and a 200GB SSD
- Applications:

App	Version	I/O library	Description
FLASH	4.4	PHDF5	2D Sedov
LAMMPS	Stable (7 Aug 2019)	MPI-IO	LJ Benchmark
QMCPACK	3.8.0	PHDF5	Moleculear H2O Test
ENZO	2.6	PHDF5	3D Collapse Test

- Comparison:
  - Score-P 6.0 with OTF2.

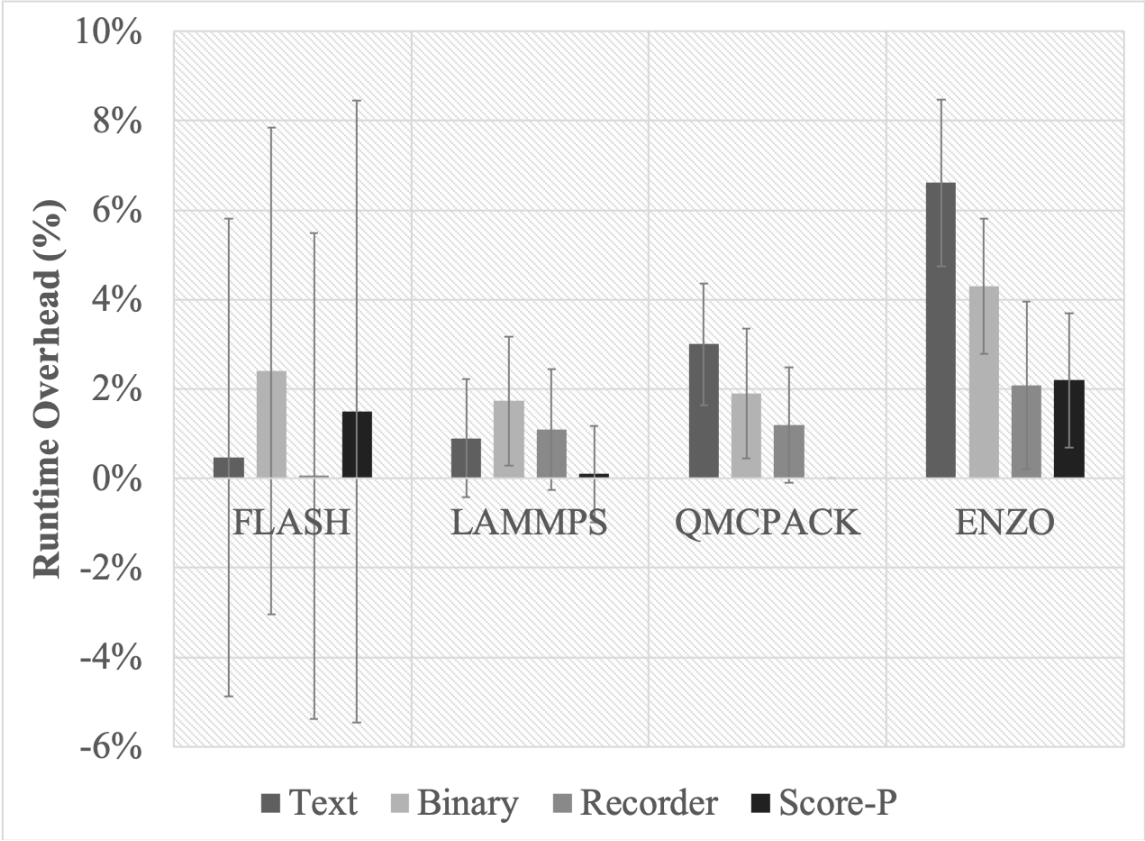
Evaluation – trace file size

- Recorder tracing format achieves at least 2x compression ratio compared to the text format.
- Recorder tracing format is able to produce similar or even small trace files yet keep more details than that of OTF2.
- The compression ratio depends on the number of repeated function calls and also the number of different arguments between two functions.



Evaluation – run time overhead

- Run time varies largely even without tracing due to the use of shared file systems.
- Measurements were repeated at least 30 times. We also show a 95% confidence interval.
- For FLASH, the variance between runs is much larger than the overhead of tracing.
- For others, Recorder with the compressed tracing format achieves similar overheads compared to Score-p



Conclusion

- Recorder is able to trace I/O function calls across multiple layers, including HDF5, MPI-IO, and POSIX.
- We implemented a Recorder-specific compact tracing format.
- We developed a set of post-processing methods and visualization routines.
- We show that in comparison with Score-p, Recorder is able to achieve similar trace file compression ratio and run time overhead yet keeping more details about the intercepted functions.