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
Urbana, IL 61801 USA
{chenw5, js39, snir}@illinois.edu

Kathryn Mohror and Elsa Gonsiorowski
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551 USA
{kathryn, gonsie}@llnl.gov

***Abstract*—Recorder is a multi-level I/O tracing tool that captures HDF5, MPI-I/O, and POSIX I/O calls. In this paper, we present a new version of Recorder that adds support for most metadata POSIX calls such as `stat`, `link`, and `rename`. We also introduce a compressed tracing format to reduce trace file size and run time overhead incurred from collecting the trace data. Moreover, we add a set of post-mortem and visualization routines to our new version of Recorder that manage the compressed trace data for users. Our experiments with four HPC applications show a file size reduction of over 2× and reduced post-processing time by 20% when using our new compressed trace file format.**

***Index Terms*—Compressed I/O Traces, Parallel I/O, MPI-I/O**

I. Introduction

The I/O subsystem of a high-performance computing (HPC) system is a critical component, because the I/O time of scientific applications can become a practical limiting factor in application throughput. In modern supercomputers, the I/O subsystem is typically multi-tiered, and includes a temporary tier such as burst buffers of fast Flash devices, a capacity tier supported by traditional disks, and a persistent tier based on tapes. Due to the complexity of these tiered systems, many strategies have been developed to ease the burden of scientific applications running on them, including I/O libraries such as HDF5 [1] and ROMIO [2], checkpointing libraries such as SCR [3] and VeloC [4], data management systems such as Data Elevator [5] and DataSpaces [6], and file systems that support temporary tiers [7]–[10].

Although these strategies make complex I/O systems much easier to use, the performance of applications using these libraries can vary dramatically when run on different I/O systems or when used with different library configuration parameters. To aid in understanding these complex I/O scenarios, we introduce our multi-level tracing tool, Recorder 2.0, that can provide key insights to end users, library developers, and file system developers. Recorder generates trace records for each layer in the I/O stack, beginning at high level I/O libraries and ending at the POSIX layer. Thus, users and library developers can see the trace of calls generated for a high level I/O operation and see how they affect performance.

Some newer file systems that support temporary tiers (e.g., UnifyFS [9] and SymphonyFS [10]) are user-level file systems that are not fully POSIX-compliant, i.e., they relax POSIX semantics in order to improve I/O performance. For these file system developers, important questions include: which functions and POSIX features do applications utilize? And to what extent can POSIX semantics be relaxed without affecting applications? To aid in answering these questions, 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. The post-mortem analysis tools can answer questions such as whether an application performs conflicting updates to the same file location.

In this paper, we present Recorder 2.0, a major update of the previous work in Recorder 1.0 [11]. The key difference between Recorder and other tracing tools is that it faithfully records all parameters of intercepted function calls. However, there were some limitations and design flaws in Recorder 1.0 that prevented users from achieving their I/O analysis goals. We summarize some of those limitations along with our solutions we implemented in Recorder 2.0 here:

- Recorder 1.0 writes trace records in plain text files that often contain redundant information for function calls with the same parameters (e.g., file name), which can result in very large trace files for I/O-heavy applications. In Recorder 2.0, we utilize a compressed trace schema that greatly reduces the overall trace file size.
- Recorder 1.0 performs I/O in a simplistic fashion - it writes trace records to the trace file without buffering which can perturb the application's performance since I/O operations can be slow. In Recorder 2.0, the logging unit caches trace records in an in-memory buffer and only writes to the trace file when the buffer is full.
- Recorder 1.0 does not capture all POSIX functions that are needed for file system developers to perform their analyses. Recorder 2.0 addresses this gap by including a more complete set of POSIX routines, now including functions such as `rename`, `unlink`, `stat`.
- To better understand the I/O behavior of HPC applications, we developed a set of analysis and visualization routines for Recorder 2.0. Those routines can perform post-mortem analysis with on-the-fly decompression.

In the rest of this paper, we refer to Recorder 2.0 as simply Recorder. We refer to the original version of Recorder as

Recorder 1.0. The rest of this paper is organized as follows. Section II discusses the existing tracing tools and explains the difference between them and Recorder. Section III describes the mechanism of Recorder and the proposed tracing format in details. We also illustrate how to work with the compressed traces by using three example analysis routines. The evaluation of Recorder is given in Section IV. And finally, conclusions and future work are given in Section V.

II. Related Work

There are many existing efforts in the area of tracing tools. Based on their scope, those tracing efforts can be classified into two categories. (1) System-level tools, e.g., iotop, iostat and blktrace, monitor system level I/O activities and report useful metrics across the entire system or at least one device or one partition. (2) Application-level tools, including Darshan [12], Score-P [13], IPM-IO [14], IOPin [15], and Recorder [11], run along with an application and only collect information about that one application. In this section, we focus on application-level tools because they are more closely related to our work.

I/O profiling tools. Darshan [12] and TAU [16] record POSIX I/O and MPI-IO activities and report statistics of individual applications. Although these tools collect their statistics with low overhead and provide a good estimation of overall performance, they do not capture the detailed information needed for in-depth analysis, e.g., function parameters and entry/exit times of function calls.

I/O tracing tools. IOPin [15], built on top of Pin [17], is a dynamic instrumentation tool for parallel I/O tracing that traces from the application layer all the way to the storage server layer. IOPin is tightly associated with the PVFS file system and does not trace I/O libraries above the MPI layer. IPM-I/O [14] extended an existing performance tool called IPM [18] to add I/O operation tracing. IPM-I/O traces POSIX I/O calls but applications need to be linked against the IPM-I/O library. VampirTrace [19] also records calls to I/O functions of the standard C library and is capable of tracing GPU accelerated applications. ScalaIOTrace [20] supports both MPI-IO and POSIX I/O and can generate compressed event logs. Score-P [13] is a popular tool suite for profiling, event tracing, and online analysis of HPC applications. It works with many other tools like TAU [16], Vampir and Scalasca [21]. In contrast to the above tools, Recorder intercepts HDF5, MPI, and POSIX I/O calls and does not require modification or recompilation of applications.

I/O trace visualization tools. MPE (MPI Parallel Environment) contains several logging and visualization tools including Upshot [22], Nupshot [23], and Jumpshot [24]. MPE focuses on MPI performance visualizations, whereas our work is tailored specifically for multi-level I/O. Other tools like Cube [25] and Vampir [26] are also able to display performance monitoring data at multiple levels, and current developments in Vampir include the analysis of application I/O behaviors. Similarly, Recorder's visualization tool can take an application's trace file as input and generate a detailed I/O report. Our work complements existing efforts because our trace files contain highly-detailed I/O information. Moreover, we are developing trace converters so that our trace files can be analyzed or visualized by other tools as well.

Trace format optimizations. The Open Trace Format Version 2 (OTF2 [27]) is an event tracing format that is used in Score-P and is highly optimized for managing large traces from parallel programs. Typically, a tool using OTF2 generates n event files for n processes along with local index files and a global definition file. The Score-P project recently added support for multi-level I/O tracing to OTF2 [28]. However, OTF2's logging API classifies I/O functions as metadata operations and data operations and ignores many details of the actual I/O functions. For example, it does not distinguish `pwrite`, `write`, or `writev`. Also, many system function calls are not intercepted such as `lstat`, `stat`, and `umask`. It is possible to manually instrument users' code to intercept those functions, but it is difficult to do so if the functions are called within shared libraries. In contrast, the trace optimizations introduced in Recorder are tailored for detailed I/O function tracing and our tool captures a wide range of POSIX and standard I/O calls.

Trace overhead reduction. Many efforts have been made to reduce tracing overhead and trace file size. OTFX [29], based on OTF2, applies filters to eliminate function calls that are shorter than a minimum duration and thus reduces intermediate memory buffer flushes. Wagner et al. [30] proposed several encoding optimizations for OTF2, including leading zero elimination and timer resolution reduction. General compression approaches have also been applied in many trace formats. CCCG (compressed complete call graphs [31]) compresses the data according to the similarity of reoccurring event sequences in a trace, within or across different processes. ScalaTrace [32] uses pattern recognition to accumulate recurring patterns to minimize trace data.

Overall, Recorder addresses key gaps in current tools support in multi-level I/O analysis. Namely, it captures detailed function tracing information (including function parameters) from a wide-range of POSIX and standard I/O function calls; it requires no code modification or recompilation; it provides analysis and visualization features specifically tailored for I/O operations; and it utilizes trace format operations that account for the detailed I/O information gathered and that reduce I/O overhead and trace file size.

III. Recorder

In this section, we describe the Recorder framework and our additions and optimizations to Recorder over the 1.0 version. While we retain the same tracing mechanism in Recorder from version 1.0, we add support for many metadata functions. Additionally, we updated Recorder's tracing format to reduce file size and tracing overhead and added post-processing support for our new compressed tracing format.

A. Framework

Understanding the I/O behavior of an HPC application is not a simple task due to the complex interactions between multiple

software components. Figure 1 shows a common parallel I/O software stack found on many current HPC systems.

In a hierarchical I/O stack, the layers are a bridge between high-level operations at the application level and low-level hardware, and offer abstractions to users that hide complex implementation details. Each layer employs optimization techniques designed to improve performance. Unfortunately, since each layer is normally treated as a black box, optimizations are seldom coordinated across layers and the source of performance bottlenecks can be extremely difficult to determine. A multi-level I/O tracing and analysis tool that presents a view of the function call flow through the entire I/O stack can expose cause and effect relationships across layers and make the origin of performance bottlenecks more apparent.

Recorder is built as a shared library so that no code modifications or re-compilations are required. Recorder uses function interposing to intercept function calls. This can be done easily in Linux systems by library preloading. Once specified as the preloaded library, Recorder intercepts HDF5, MPI, and POSIX function calls issued by the application and reroutes them to the tracing implementation. During the tracing process, Recorder first saves the function name, file name, function entry timestamp, and all function parameters. Next, Recorder calls the original library function and records its exit timestamp. Recorder then constructs and compresses a trace record that stores all this information. This whole process is depicted with an example in Figure 2. Details about encoding and compression will be given in Section III-B.

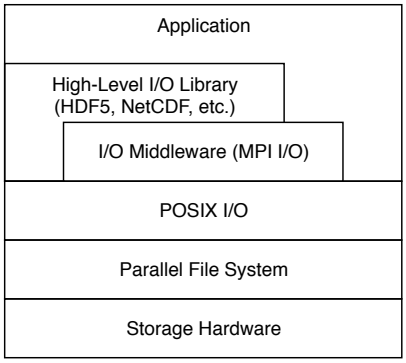

Fig. 1. Parallel I/O Software Stack

B. Recorder Tracing Format

In Recorder 1.0, all outputs are in ASCII format as it is human readable and good for debugging purposes. However, this text format retains redundant information and can easily result in very large trace files. The run time overhead and post-processing time can be high in Recorder 1.0 due to the I/O time incurred in writing and analyzing large trace files.

One straightforward way to solve this problem is to compress trace records at run time or afterwards using an external compression library like zlib [33]. We implemented this method in Recorder as our first approach. However, most external compression libraries use general algorithms

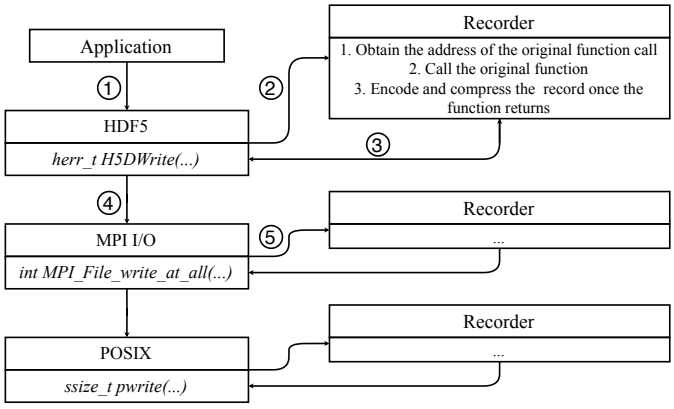

Fig. 2. Example of instrumentation of the I/O stack by Recorder. ① Application calls the HDF5 library method `H5Dwrite`. ② Recorder intercepts the function and performs the tracing process. ③ Recorder calls the real `H5Dwrite` function. ④ `H5Dwrite` calls the MPI function `MPI_File_write_at_all`. ⑤ `MPI_File_write_at_all` is also intercepted and recorded by Recorder. This continues until the I/O stack reaches the POSIX layer.

that work with arbitrary strings, which means it is difficult to perform record-level selective decompression during post-processing since the general compression/decompression algorithms are unaware of the structure information of the trace record. Selective decompression is important, as many analyses need only a subset of the record fields. Moreover, we noticed that with online zlib compression, the overall overhead is similar to that incurred with the text format: The compression/decompression overheads are similar to the I/O time saved by writing and reading shorter files. Thus, we pursued the development of a Recorder-specific tracing format and a format-aware compressing algorithm.

In Figure 3 we show our compact binary encoding for trace records. The status byte of the record is used to indicate whether the current record is compressed, which we will discuss later. The Δtstart and Δtend are calculated as follows:

$$\Delta tstart = (T_s - T_0)/T_R$$

$$\Delta tend = (T_e - T_0)/T_R$$

T_s and T_e are the function entry and exit time. T_0 is the starting timestamp of the program, and T_R is the adjustable time resolution. The 10th byte, func_id is a non-negative integer representing the corresponding function. We created a function table where each function intercepted by Recorder has a hard-coded entry for its name and identifier. The rest of the bytes in the record are variable length function arguments, which we simply write in string characters. One exception is that file names are mapped into integers since they are often long strings and appear repeatedly in argument lists.

Recorder uses a global metadata file to keep program level information including time resolution, program starting time, and user-specified options. In addition, each MPI rank also keeps a local metadata file to keep track of <filename, integer id> mappings and several per-rank metrics such as function counters and file access counters.

status	Δtstart	Δtend	func_id	arg_1	...	arg_n
1Byte	4Bytes	4Bytes	1Byte	variable		

Fig. 3. Recorder Binary Encoding of Trace Records

Although the binary encoding saves space for long strings within a record, there are still many repetitions between records. For instance, an application may call `MPI_Allreduce` many times in sequence to collect data values. Or an application may call `MPI_File_write_at` to write a checkpoint file, which in turn calls `pwrite` multiple times to perform the actual job. To take advantage of this temporal locality of function calls, we use a peephole-based compression technique.

The compression technique keeps a sliding window of a few of the most recent records. Once a new record arrives, the algorithm checks if the new record is similar to one of the records in the sliding window. The similarity is decided using two criteria: (i) two records must store the same function, (ii) this function should contain at least one argument and the two records should have at least one argument in common. The new record is compressible if we can find a similar record in the sliding window. If multiple similar records exist, then the algorithm prefers the one with most common arguments. Ties are broken by choosing the most recent similar record. If the new record is not compressible, we simply encode it as before (the binary encoding shown in Figure 3). However, if a similar record exists in the sliding window, then we compress the new record by only keeping the differences.

The encoding of compressed records is shown in Figure 4. It is similar to the uncompressed binary encoding except that the func_id is now ref_id that stores the relative location of that similar record. We only need one byte for the ref_id as the sliding window is small (3 by default). For uncompressed records, the eight status bits are all zeros, whereas for a compressed record, the first status bit is always 1, and the following 7 bits are used to indicate the indices of the arguments that are different. It is worth noting that this representation only supports compression of functions with up to 7 arguments. However, this is not a major limitation in practice because it is adequate to handle commonly-used functions like `pwrite`, `MPI_Send`, and `MPI_Recv`. Figures 5 and 6 give examples of records before and after compression. Here a window of 3 recent records is capable of compressing all functions after their first appearance.

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

Fig. 4. Recorder Compressed Encoding

C. Analyzing routines

We developed a set of helper functions to interact with the compressed traces and a converter that translates the traces to human-readable format. We developed functions that perform helper operations, denoted here by `DECOMPRESS()` that decompresses one record, `LOAD()` that reads one field (e.g., status, func_id, etc.) from a decompressed record, and `COMPRESSED()` that checks if a record is compressed. These helper functions are responsible for reading and decompressing the trace files on-the-fly and greatly simplify the logic of analysis task development.

We use three examples to show how analysis can be performed efficiently on the compressed traces.

1) Function counters: As mentioned earlier, each rank maintains a function counter along with several other per-rank metrics in their local metadata file. This metadata file is normally very small so no compression or special encoding is used. Users simply need to perform a reduction operation on the counters collected by each rank to compute this metric.

2) Statistics: Application-level statistics are important for understanding overall application I/O performance. For example, I/O bandwidth is a key metric for I/O intensive applications: one can check read bandwidth to determine if I/O is a bottleneck. To report application-level statistics, we need to aggregate information from each rank. Algorithm 1 gives the pseudocode for computing average read bandwidth. We examine and decompress each trace record only if it contains a specific target function, e.g., `pread` or `read`. To achieve this, we load and check func_id (line 6-10) before the decompression at line 12.

The time complexity of this algorithm is $O(N)$, where N is the total number of records across all ranks. The same method will apply to many other statistics as well. Depending on the specific analysis goals, users may need to change the function set at line 10 and the fields loaded at line 15.

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
```

3) Check for overlapping I/O: A more difficult task, and also one of the motivations of this work, is to check for overlapping I/O operations: Does a process ever write to the

tstart	tend	funciton	args					
1572448298.799995	1572448298.799995	MPI_Comm_rank	MPI_COMM_WORLD		0x7ffdc64a6f64			
1572448298.800000	1572448298.800000	MPI_Comm_rank	MPI_COMM_WORLD		0x7ffdc64a6f64			
1572448298.800002	1572448298.800002	MPI_Comm_rank	MPI_COMM_WORLD		0x7ffdc64a6f64			
1572448298.800004	1572448298.800004	MPI_Comm_rank	MPI_COMM_WORLD		0x7ffdc64a6f64			
1572448298.802099	1572448298.802102	MPI_Allreduce	0x564591831ec0	0x5645917a10a0	11	MPI_DOUBLE_PRECISION	1476395010	MPI_COMM_WORLD
1572448298.802106	1572448298.802110	MPI_Allreduce	0x564591831f20	0x564591831e60	11	MPI_DOUBLE_PRECISION	1476395009	MPI_COMM_WORLD
1572448298.802955	1572448298.802960	MPI_File_write_at	0x5645916fdaa0	39416	0x5645917a1cf0	256	MPI_BYTE	0x7ffdc649cc70
1572448298.802957	1572448298.802958	pwrite	./sedov_check.h5		0x5645917a1cf0	256		39416
1572448298.802960	1572448298.802962	MPI_File_write_at	0x5645916fdaa0	34184	0x7ffdc64a73b0	48	MPI_BYTE	0x7ffdc649c430
1572448298.802965	1572448298.802968	pwrite	./sedov_check.h5		0x7ffdc64a73b0	48		34184

Fig. 5. Example records before encoding and compression, in total of 10 functions and 40 arguments.

status	Δtstart	Δtend	func_id/ref_id	args / diff_args					
B'00000000	0	0	100	1 0x7ffdc64a6f64					
B'10000000	5	5	-1						
B'10000000	7	7	-1						
B'10000000	9	9	-1						
B'00000000	2095	2098	116	0x564591831ec0	0x5645917a10a0	11	2	1476395010	1
B'11100100	2102	2106	-1	0x564591831f20	0x564591831e60			1476395009	
B'00000000	2951	2956	92	0x5645916fdaa0	39416	0x5645917a1cf0	0x7ffdc649cc70	256	3
B'00000000	2953	2954	11	0	0x5645917a1cf0		256	39416	
B'10111010	2956	2958	-2	34184	0x7ffdc64a73b0		48	0x7ffdc649c430	
B'10111000	2961	2964	-2	0x7ffdc64a73b0	48		34184		

Fig. 6. Example records after encoding and compression. With a sliding window of 3 records, every function can be compressed after their first appearance.

same part of a file more than once? Do multiple processes write or read the same part of a file? Do applications perform read after write or write after read to the same offset? These questions were sparked by the design of a lightweight user level file system, UnifyFS [9]. The answers to these questions could help file system developers make better optimization decisions by having a better understanding of the I/O behaviors of users' applications.

We denote the start offset and end offset of an I/O function as os and oe. And we use a table $P[r_1, r_2]$ to keep track of the existence of overlapping I/O patterns between rank r_1 and r_2, where r_1 and r_2 are possibly equal. Each entry in the table stores the patterns we found, including read after read (RAR), write after write (WAW), read after write (RAW), and write after read (WAR). Similar to algorithm 1, we iterate through every record of every rank to build a list of tuples T. Each tuple has the form of $(t, r, os, oe, read)$, where t and r are the timestamp and rank of the record, and $read$ is a binary value indicates whether this is a read function, e.g., `pread`, `read`, or `readv`. We use algorithm 2 to fill in the table P. Note that we only show how to identify the RAW and WAR patterns for table P due to space restrictions. RAR and WAW patterns can be identified similarly.

In the worst case where all I/O operations access the same range, the table has $\Theta(N^2)$ entries and any algorithm will take $\Theta(N^2)$ time to finish. However, in most common cases where most accesses do not overlap with each other, or when one only needs to decide whether an overlap exists, this algorithm takes $\Theta(N \log N)$ due to the sorting phase.

Algorithm 2 Check for overlapping I/O patterns

```
1:  Sort tuples by os
2:  for each tuple T_i do
3:      for each tuple T_j after T_i do
4:          if os_j > oe_i then
5:              break  ▷ The remaining tuples cannot overlap with T_i
6:          else                    ▷ T_i and T_j are overlapped
7:              if read_i && !read_j then
8:                  if t_1 < t_2 then
9:                      P[r_i, r_j].WAR ← 1
10:                 else
11:                     P[r_i, r_j].RAW ← 1
12:             Similarly, check for RAR and WAW
```

IV. Evaluation

Our experiments were performed on Stampede2 at TACC. The Stampede2 system is configured with 4204 Dell/Intel Knights Landing (KNL) nodes and 1736 Intel Xeon Skylake (SKX) nodes. Each SKX node (Intel Xeon Platinum 8160) includes 48 cores, 192GB DDR-4 memory, and a 200GB SSD. Compute nodes have access to a dedicated Lustre Parallel file system provided by Cray. An Intel Omni-Path Architecture switch fabric connects the nodes and storage through a fat-tree topology with a point-to-point bandwidth of 100 Gb/s (unidirectional speed). We used 24 SKX nodes with 24 MPI ranks per node in all our experiments.

TABLE I
CONFIGURATIONS OF EACH APPLICATION

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

We evaluated Recorder and compared it with Score-P 6.0 using four HPC applications: FLASH [34], LAMMPS [35], QMCPACK [36], and ENZO [37]. The applications are compiled with Intel MPI library 17.0.3, Intel Math library 17.0.4, and Parallel HDF5 (PHDF5) library 1.8.16. The configurations of each application are summarized in Table I. We run a certain number of steps (1,000 steps for LAMMPS and FLASH, 2,000 steps for QMCPACK, and 0.5 seconds simulation time for ENZO) and write out a total of 5 checkpoints for each application. The different set up is chosen to keep the run time of each application manageable and relatively the same.

We configured Recorder and Score-P to trace all MPI and POSIX function calls, and used Score-P's filtering mechanism to ignore user function calls to make sure that Recorder and Score-P record the same set of function calls. We disabled the profiling functionality of Score-P so we could have a fair comparison of run time overhead.

For the run time overhead experiments, we repeated measurements at least 30 times for each application and reported averages and a 95% confidence interval. The result is shown in Figure 7. In the figure, Text, Binary, and Recorder denote three different output formats as we discussed in Section III: Text is the text format with ASCII encoding, Binary is the binary format without compression, and Recorder represents our compressed encoding schema.

The overhead experiments show a high degree of variability for FLASH between runs. The noise is largely introduced by the parallel I/O phase due to the use of the shared parallel file system and is present even without tracing. The variance between runs is much larger than the overhead of tracing, so that the overhead of tracing cannot be estimated with high confidence. Also note that we do not show the overhead of Score-P for QMCPACK as it runs about 3 times slower with Score-P. Overall, when using the compressed tracing format, Recorder achieves similar or smaller overheads Score-P.

Figure 8 shows the aggregated trace file sizes of the different encoding methods, each normalized to the size of the text format. Both OTF2 and the Recorder format achieve over $2\times$ compression ratios in all four applications. Note that Recorder keeps more information per function call (it records all function arguments).

The traces of QMCPACK and ENZO had slightly better compression ratios than the other two applications. The final compression ratio depends on how many records can be compressed, how many arguments within a record can be compressed, and the length of compressible arguments. To understand the difference of compression ratios in these four applications, we show in Table II and Table III the number of

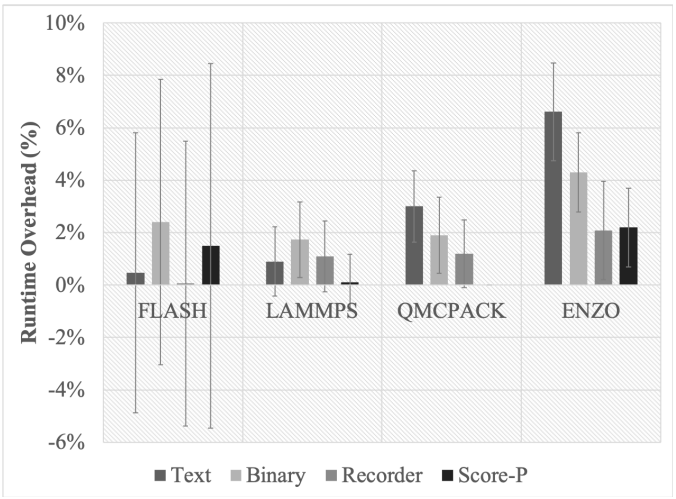

Fig. 7. Run time overhead of Recorder and Score-P with 95% confidence intervals

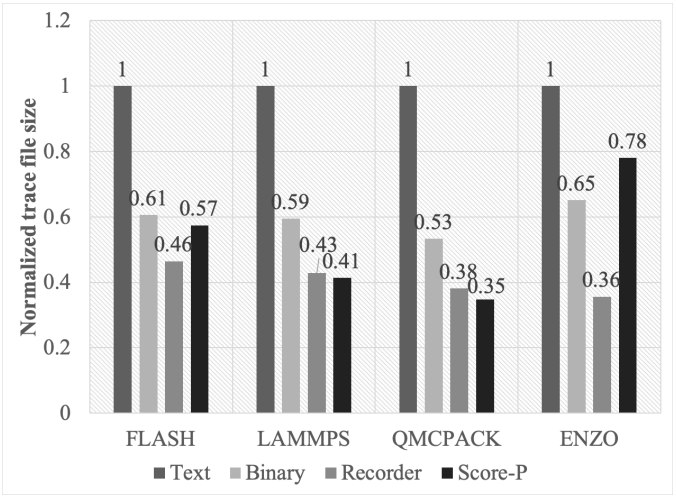

Fig. 8. Aggregated trace file size with different encoding (normalized to text format)

compressed records and the number of compressed arguments. QMCPACK has the lowest percentage of compressible records but within each compressible record, 70% of its arguments are compressed, i.e. do not need to be stored. On the other hand, 96% of records in ENZO can be compressed but 47% arguments of those compressible records still need to be kept.

ENZO has a surprisingly high percentage of compressible records. We use the visualization routines built into Recorder to show the number of records kept by each rank in Figure 9

TABLE II
NUMBER OF COMPRESSED RECORDS OF EACH APPLICATION

App	# Records	# Compressed records	Ratio
FLASH	16,612,758	8,443,078	50.82%
LAMMPS	18,628,201	8,696,241	46.68%
QMCPACK	12,944,238	5,402,263	41.73%
ENZO	338,803,012	325,794,888	96.16%

TABLE III
NUMBER OF COMPRESSED ARGUMENTS OF EACH APPLICATION

App	# Arguments	# Compressed arguments	Ratio
FLASH	40,913,405	26,628,876	65.85%
LAMMPS	40,138,951	24,577,733	61.23%
QMCPACK	21,251,485	15,028,209	70.72%
ENZO	2,176,397,368	1,158,611,935	53.24%

and the top 10 visited functions in Figure 10. These 10 most-visited functions account for over 99% of all function calls. As we can see, 9 of them are repeatedly-invoked MPI functions, which makes them very easy to compress.

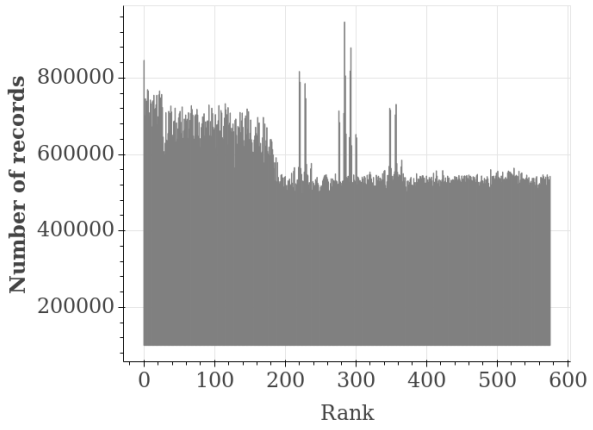

Fig. 9. Record count of each MPI rank in ENZO

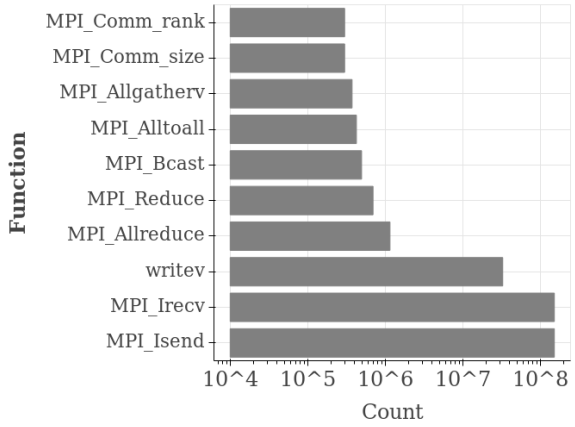

Fig. 10. The 10 most-visited functions in ENZO

Finally, we evaluated the algorithm for detecting overlapping I/O operations described in Section III-C with FLASH traces containing four million records. We conducted our experiment on a local workstation, equipped with an AMD Ryzen 7 Eight-Core CPU and 8GB DDR-4 memory. Figure 11 compares the execution time of the different phases of the algorithm for three different encodings: the uncompressed binary encoding (first bar), our compressed encoding (second bar), and our compressed encoding with selective decompression (third bar). With the binary format, since no records are compressed, there is no decompression cost. In contrast, with the Recorder format, there is a cost to decompression but also an overall improvement in "Read & Decode" time from reading and decoding smaller trace files. Moreover, with selective decompression, the algorithm only decompresses and keeps in memory the needed functions and fields. Thus, selective decompression requires more time to build the interval list as one needs to check and decompress functions in this step, but this also results in a smaller memory footprint and potentially better cache localities. Note that for large trace files such as for ENZO, with a total of 24GB in our experiments, it may not be possible to fit all records in memory on a single machine, so a selective loading or decompression method would have to be used for this analysis. The costs of the sorting and testing phases are similar for all three encodings, since the constructed interval lists will be the same in all three cases. Overall, using our Recorder format can improve detection of overlapping I/O operations by 15.75% and with selective decompression, the performance is improved by about 20%.

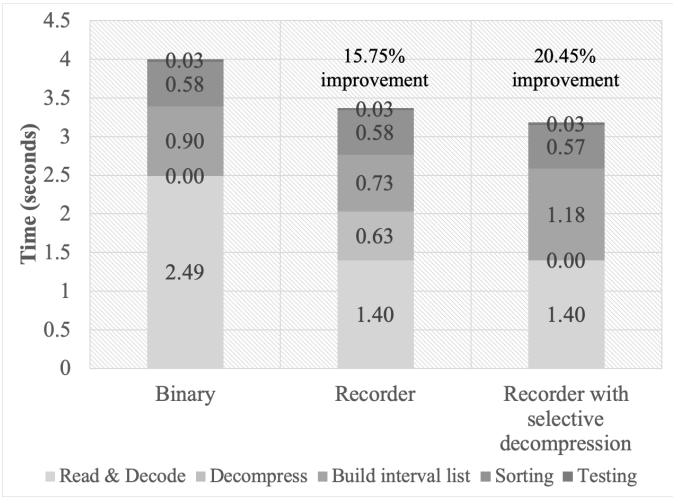

Fig. 11. Time decomposition for checking overlapping patterns in FLASH

V. CONCLUSION AND FUTURE WORK

In this paper, we described Recorder 2.0, a multi-level tracing tool for HPC applications. Recorder 2.0 is a major update of the first version of Recorder with many new features. Recorder 2.0 uses a compressed encoding that achieves over 2× reduction in trace file sizes and includes a set of post-processing and visualization routines for better understanding the I/O behavior of HPC applications. We have made Recorder 2.0 publicly available at https://github.com/uiuc-hpc/Recorder.

In our future work, we plan to support other high level I/O libraries, such as NetCDF, and also support manual instrumentation of specific user routines.

ACKNOWLEDGMENT

This research was partly supported by NSF SHF award 1763540 and was performed under the auspices of the U.S.

Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-802986).

This work used the Extreme Science and Engineering Discovery Environment (XSEDE) Stampede2 at the Texas Advanced Computing Center (TACC) through allocation TG-CCR130058.

REFERENCES

[1] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An Overview of the HDF5 Technology Suite and Its Applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, ser. AD '11, 2011.

[2] R. Thakur, W. Gropp, and E. Lusk, "On Implementing MPI-IO Portably and with High Performance," in *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, ser. IOPADS '99, 1999.

[3] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinksi, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10, LLNL-CONF-427742*, November 2010.

[4] 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)*, May 2019.

[5] B. Dong, S. Byna, K. Wu, Prabhat, H. Johansen, J. N. Johnson, and N. Keen, "Data Elevator: Low-Contention Data Movement in Hierarchical Storage System," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, Dec 2016.

[6] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows," *Cluster Computing*, Jun 2012.

[7] J. Xu and S. Swanson, "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories," in *14th USENIX Conference on File and Storage Technologies ({FAST} 16)*, 2016, pp. 323–338.

[8] T. Wang, W. Yu, K. Sato, A. Moody, and K. Mohror, "BurstFS: A Distributed Burst Buffer File System for Scientific Applications," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2016.

[9] L. L. N. Laboratory, "UnifyFS: A Distributed Burst Buffer File System," Dec. 2019. [Online]. Available: https://github.com/LLNL/UnifyFS

[10] S. Oral, S. S. Vazhkudai, F. Wang, C. Zimmer, C. Brumgard, J. Hanley, G. Markomanolis, R. Miller, D. Leverman, S. Atchley *et al.*, "End-to-end I/O portfolio for the Summit Supercomputing Ecosystem," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2019, p. 63.

[11] H. Luu, B. Behzad, R. Aydt, and M. Winslett, "A Multi-level Approach for Understanding I/O Activity in HPC Applications," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2013, pp. 1–5.

[12] 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*. IEEE, 2009, pp. 1–10.

[13] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony *et al.*, "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.

[14] A. Uselton, M. Howison, N. J. Wright, D. Skinner, N. Keen, J. Shalf, K. L. Karavanic, and L. Oliker, "Parallel I/O Performance: From Events to Ensembles," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–11.

[15] S. J. Kim, S. W. Son, W.-k. Liao, M. Kandemir, R. Thakur, and A. Choudhary, "IOPin: Runtime Profiling of Parallel I/O in HPC Systems," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012, pp. 18–23.

[16] S. Shende, A. D. Malony, W. Spear, and K. Schuchardt, "Characterizing I/O Performance Using the TAU Performance System," in *PARCO*, 2011, pp. 647–655.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[18] D. Skinner, "Integrated Performance Monitoring: A Portable Profiling Infrastructure for Parallel Applications," in *Proc. ISC2005: International Supercomputing Conference, Heidelberg, Germany*, 2005.

[19] M. Jurenz, R. Brendel, A. Knüpfer, M. Müller, and W. E. Nagel, "Memory Allocation Tracing with VampirTrace," in *International Conference on Computational Science*. Springer, 2007, pp. 839–846.

[20] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, "Scalable I/O Tracing and Analysis," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*. ACM, 2009, pp. 26–31.

[21] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[22] V. Herrarte and E. Lusk, "Studying Parallel Program Behavior with Upshot," Argonne National Laboratory, Tech. Rep. ANL–91/15, 1991.

[23] E. Karrels and E. Lusk, "Performance Analysis of MPI Programs," in *Proceedings of the Workshop on Environments and Tools For Parallel Scientific Computing*, J. Dongarra and B. Tourancheau, Eds. SIAM Publications, 1994, pp. 195–200.

[24] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward Scalable Performance Visualization with Jumpshot," *The International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 277–288, 1999.

[25] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr, "Cube v4: From Performance Report Explorer to Performance Analysis Tool," *Procedia Computer Science*, vol. 51, pp. 1343–1352, Jun. 2015.

[26] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir Performance Analysis Tool-Set," in *Tools for high performance computing*. Springer, 2008, pp. 139–155.

[27] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf, "Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries," in *PARCO*, vol. 22, 2011, pp. 481–490.

[28] R. Tschüter, C. Herold, B. Wesarg, and M. Weber, "A Methodology for Performance Analysis of Applications Using Multi-layer I/O," in *European Conference on Parallel Processing*. Springer, 2018, pp. 16–30.

[29] M. Wagner, J. Doleschal, and A. Knüpfer, "MPI-focused Tracing with OTFX: An MPI-aware In-memory Event Tracing Extension to the Open Trace Format 2," in *Proceedings of the 22nd European MPI Users' Group Meeting*. ACM, 2015, p. 7.

[30] M. Wagner, A. Knupfer, and W. E. Nagel, "Enhanced Encoding Techniques for the Open Trace Format 2," *Procedia Computer Science*, vol. 9, pp. 1979–1987, 2012.

[31] A. Knüpfer and W. E. Nagel, "Compressible memory data structures for event-based trace analysis," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 359–368, 2006.

[32] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. De Supinski, "ScalaTrace: Scalable Compression and Replay of Communication Traces for High Performance Computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 8, pp. 696–710, 2009.

[33] J. loup Gailly and M. Adler, "zlib A Massively Spiffy Yet Delicately Unobtrusive Compression Library," Dec. 2017. [Online]. Available: https://www.zlib.net/

[34] "Flash Center for Computational Science," 2019. [Online]. Available: http://flash.uchicago.edu/site/flashcode

[35] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *Journal of computational physics*, vol. 117, no. 1, pp. 1–19, 1995.

[36] J. Kim, A. D. Baczewski, T. D. Beaudet, A. Benali, M. C. Bennett, M. A. Berrill, N. S. Blunt, E. J. L. Borda, M. Casula, D. M. Ceperley *et al.*, "QMCPACK: an open source *ab initio* quantum Monte Carlo package for the electronic structure of atoms, molecules and solids," *Journal of Physics: Condensed Matter*, vol. 30, no. 19, p. 195901, 2018.

[37] G. L. Bryan, M. L. Norman, B. W. O'Shea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman *et al.*, "Enzo: an adaptive mesh refinement code for astrophysics," *The Astrophysical Journal Supplement Series*, vol. 211, no. 2, p. 19, 2014.