Automatic Generation of I/O Kernels for HPC Applications

Babak Behzad* Hoang-Vu Dang* Farah Hariri* Weizhe Zhang† Marc Snir*‡
*University of Illinois at Urbana-Champaign
†Harbin Institute of Technology ‡Argonne National Laboratory

***Abstract*—The study of the I/O performance of a parallel application can be facilitated by the use of an *I/O kernel* – a program that generates the same I/O calls as the original application, but can be executed much faster. Such I/O kernels are especially important when the programs under study are proprietary or classified, and only available in binary form.**

In this paper, we show how to create automatically such an I/O kernel, by executing the target application with an instrumented I/O library, next "compressing" the resulting I/O traces into a compact C program that generates those traces.

***Keywords*—Parallel I/O, I/O trace and replay, I/O kernels**

I. Introduction

I/O can be a significant bottleneck on HPC application performance. The need to increase checkpoint frequency and the increasing emphasis on big data analytics increase the importance of I/O. Parallel I/O systems are complex: I/O is often done at the application level using a high-level library, such as HDF5 [9]; HDF5 is implemented atop MPI-IO [8] which, in turn, performs POSIX I/O calls against a parallel file system, such as Lustre [17]. Each of these subsystems has multiple configuration knobs and performance can be very sensitive to their settings.

Efforts conducted towards improving I/O performance of current HPC platforms with their large amount of parallelism can be categorized into different categories such as algorithm designs like data sieving and collective I/O [18], manual optimizations of applications based on I/O expertise [11], and autotuning to obtain a good I/O configuration [4]. All of these techniques can be assisted by I/O profiling and tracing.

I/O Profiling tools characterize I/O performance of HPC applications by counting I/O-related events. Among those, one can mention Darshan [7] and IOPin [12] and among the tracing tools Tracefs [2], //TRACE [15], ScalaHTrace [20], RIOT-IO [19]. Tracing tools log the timing of each I/O function calls, their arguments and their return values, etc. Profiling tools are less intrusive and are useful in identifying potential I/O bottlenecks in the performance; but not enough information is gathered for a detailed understanding of I/O. Section II describes each of these tools and their difference with this work in detail.

Realistic I/O kernels are an important tool for the study of I/O. They can be used to evaluate storage systems (both current storage systems, through execution and new designs, through simulation); they facilitate collaboration between institutions: There are many cases where full-fledged application codes can not be shared between institutions, because of their proprietary or classified nature. An I/O kernel can provide detailed information on the I/O characteristics of such an application while providing little information on the computation it performs. In addition, it can be run faster. I/O kernels are typically built manually, which is a time-consuming and error-prone process. An alternative is to record a trace of the I/O operations and "replay" the trace. Existing systems capture traces mostly at the level of POSIX I/O and MPI-IO calls [15], [2], [20]. Ganger [10] explains the limitations of this approach.

Previously, we implemented a multi-level I/O tracing framework, called Recorder [14]. It captures I/O function calls at multiple levels of the parallel I/O stack, including HDF5, MPI-IO, and POSIX I/O. Having such tracing framework enabled us to compare and contrast the abilities to replay and/or generate I/O kernels from the traces at every level. We have concluded that HDF5 library eases the process of creating a standalone parallel I/O kernel significantly. Every object in HDF5 such as files, dataspaces, datasets, attributes, groups, etc. have a unique integer identifier. Therefore, it is easy to keep track of them in the generated code. Additionally, most of the HDF5 I/O operations have to be called collectively. This eases the process of merging traces across the ranks. Since the HDF5 calls determine the function calls at the lower levels, capturing only these calls causes no loss of information.

Our current work (first introduced in our work-in-progress at [3]) is focused on automating the creation of an *I/O kernel code*: A code that generates the same (HDF5) I/O calls as the original program, while shedding details of the computation. Furthermore, we wish to do so without requiring access to the program's source. This is because in many situations, we can run the program having only access to binaries.

We start with trace files generated by an instrumented HDF5 library, i.e. the Recorder at each process; the traces are merged into one file; the order of the I/O operations is preserved by merging these traces correctly. The I/O kernel is generated from this merged trace file.

A naive application of this algorithm would yield a kernel program of length proportional to the total length of the trace files. However, simple pattern matching based compression techniques can be used to reduce the size of the kernel code: It is often possible to generate a kernel code of length proportional to the number of HDF5 calls in the original code.

The remainder of this paper is organized as follows: We discuss related work in the next section and discuss our framework in Section III. Section IV presents the results of

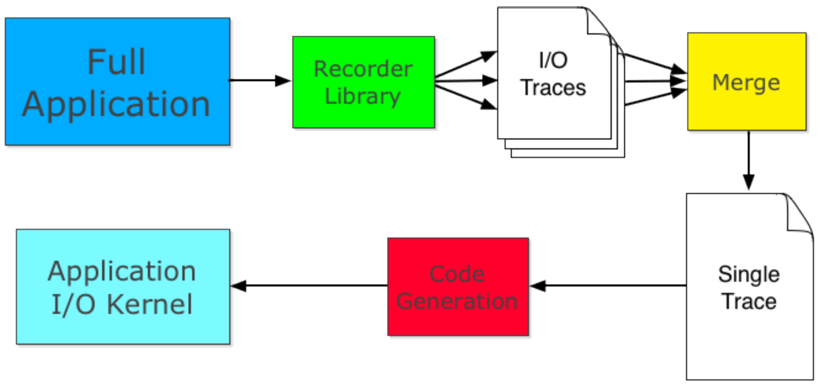


Fig. 1. Flow of the Framework

experiments and Section V provides conclusions and discusses the future work.

II. Related Work

Tracefs[2] is a low-overhead and flexible tracing file system that intercepts operations at the VFS level. The traces are recorded and useful for security and debugging.

//Trace [15] provides a detailed framework on POSIX-Level I/O recording and replaying. It puts more emphasis on the average replay accuracy of the parallel replayer, which can mimic the behavior of the traced application. Inter-node data dependencies and computing times are discovered in order to create more representative workloads for storage systems evaluation.

Scala-HTrace [20] focuses on the recording and compression of MPI-I/O level traces. It utilizes histograms based on a user-specified merge precision level, which replays statistical histogram traces without decompressing the original trace file.

Our work is distinct in that it traces I/O at the level of the HDF5 application calls. Since MPI-IO and file system activities result from the HDF5 calls, tracing at the highest possible level provides a complete view of I/O activities. In addition, we focus on lossless compression of the traces, so that no information is lost.

Skel [13] is probably the closest work to this work; Both with the same objective, but with very different approaches. Skel creates I/O skeletal applications by utilizing the ADIOS [1] framework. ADIOS users configure the I/O of the applications by creating an external XML file. Skel makes use of this XML configuration and an additional XML file for the parameters to create skeletal I/O application. In our work, we do not need any configuration file and running the application is enough to get the traces and generate the I/O kernel. Additionally, our approach replays all the HDF5 I/O calls of an application leading to the exact same I/O behavior of the original applications.

III. Framework

This section introduces our framework for automatically generating I/O kernels from HPC applications. Figure 1 shows the overall flow of this framework. An HPC application is first linked to our recorder library. The recorder library stores traces for all the I/O calls into a separate trace file for each MPI rank. Therefore, after running the application, n trace files are generated, where n is the number of MPI processes. These n trace files are fed into another tool of our framework in order to be merged together. Once these traces are merged into a single trace file, a code generator generates an SPMD MPI-based application for it. Each of these three steps are explained in detail in the following subsections:

A. *I/O Tracing: Recorder*

Figure 2 shows the process of intercepting an HDF5 function call (`H5Fcreate()`, used for creating an HDF5 file). Once Recorder is used, it intercepts HDF5 function calls issued by the application and reroutes them to the tracing implementation where the timestamp, function name, and function parameters are recorded. The original HDF5 function is called after this recording process. Once the program returns from the HDF5 call, return value, and the duration of the call are also recorded. This tracing approach is transparent to the user because alterations are made without change to application or library source code. Recorder can be built as a shared library and linked to the application at runtime so that it does not require modification or recompilation of the application. It also can be built statically using `-wrap` functionality. As mentioned in [14] the overhead of this mechanism is negligible in both cases.

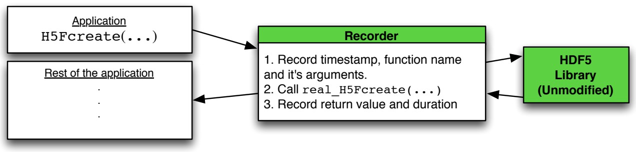

Application
H5Fcreate(...)
Rest of the application
Recorder
1. Record timestamp, function name and it's arguments.
2. Call real_H5Fcreate(...)
3. Record return value and duration
HDF5 Library (Unmodified)

Fig. 2. Method of intercepting HDF5 calls by the Recorder

B. Trace Merging

The output of the recorder is n log files, where n is the number of MPI processes. Before executing the merging process, these files get coded into a uniform format that is easier to process. The merger takes as input these log files, and outputs one file containing the merged traces. We expect that many HDF5 calls are collective, therefore, the merge process attempts to identify tuples of records, one from each file, that have the same "signature" (we discuss how to match signature later). We assume that matching records will appear at roughly the same location in the different files (i.e., that the I/O operations executed by different processes are similar). Therefore we keep, within each trace file, a fixed-size window of records under consideration for merge. Note that incomplete matching affects the size of the merged trace, but not its correctness. Also note that we are not relying on any timing mechanism such as NTP or timestamps for merging the I/O operations; we simply rely on the order of I/O operation called by the application.

We keep track of our position in each file t^i using pointers p^i. If all the records we are currently pointing at have the same signature, we merge them. If not, we pick from the current tuple of records the record that has the farthest matches in the other files and move it to the merged file. The intuition behind this heuristic is that we would like to merge traces in a greedy manner, merging the largest number of records at each time. Hence, if a record has a match in another file that is close by, we prefer to keep it until we can merge it with its matches. This case is illustrated in Figure 3.

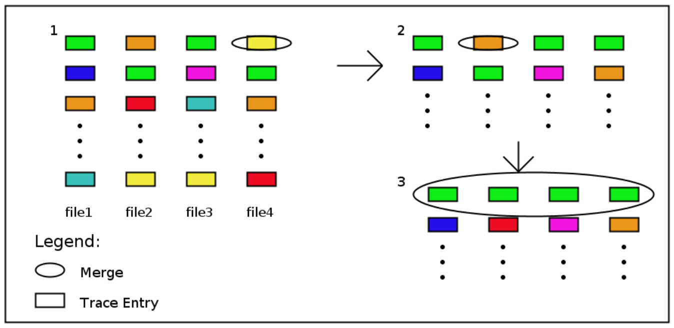


Fig. 3. Illustrating three consecutive merging operations

In order to quickly find the matches of a given record, a hash table keeps track of all the records from all the files within a windows of m records consecutive records. During our experiment we fixed this value at a constant as 200. In order to understand our data structure, we define the following terms:

- Let n be the number of MPI processes.
- Let e be a record in a trace file. It has the function name, arguments and other recorded information.
- The signature of a record e is defined by a function $k(e)$. Signatures are chosen so that records have the same signature if and only if they *match* and can be merged. Matching records have the same function name and same values for *significant parameters*. The set of significant parameters is different from function to function and are tagged at a trace preprocessing step. An example of a significant argument is the MPI communicator since the difference in the communicator clearly indicates the two traces come from different MPI calls. An example of non-significant argument is pointer addresses since they are always different from function to function.
- Let $\delta(e)$ be the smallest distance to a matching record of e in another file.

The hash table uses a hash of $k(e)$ to store records. For each value of $k(e)$, we maintain a linked list of records with that signature within the window for each MPI process. The records in the linked list are ordered in an order consistent with their order in the trace files. We also store, with each record, the value of δ which represents the current value of $\delta(e)$ within the window. Assuming adding, removing or searching for an entry from the hash table is constant time, we could keep update the δ when an entry is added or removed from the table in constant time. Similarly whenever we need $\delta(e)$, we query δ in the hash table of $k(e)$, and thus it is constant time.

The algorithm considers the current records on each file and decide to merge only if they are all matching each other. Otherwise, we pick the record e with the largest value $\delta(e)$ and move it to the merged file.

Algorithm 1 shows the high level pseudocode of the merging algorithm. First, we initialize all the pointers p^is to zero referring to the first entry of each log file. Then, we insert the first m entries of each processors into our hash table h. While we did not reach the end of all the trace files, we keep repeating the merging process which executes the two cases previously explained.

Algorithm 1 Merging Algorithm

```
Input: Trace files to be merged
Output: File containing the merged trace (OF)
Variables:
$p^i$= current position in the ith trace file
$m$ = maximum depth
$n$ = total number of trace files
$t^i$ = ith trace file
$0 \leq i \leq n$
$h$ = the hash-table data structure
Pseudo Code:
Initialize all $p_i$ s to zero.
Insert the first $m$ entries of each $t^i$ to $h$.
while $\exists p^i != t^i.size()$ do
    if all events pointed to by $p^i$s are matched then
        mergeEvents($e_{p^i}$ for any $i$).
    else
        $\forall p^i$, $\delta_i$ = getDistance($e_{p^i}$)
        $\delta_{max}$ = max($\delta_i$)
        mergeEvents($e_{p^j}$ for $j$ corresponding to $\delta_{max}$).
    end if
end while
```

To understand our algorithm complexity, let E be the total number of events. In general cases, for each event e, it will

Algorithm 2 mergeEvent method

```
Input: Trace event $e$
Pseudo Code:
$item$ = $h(k(e))$
Pop and merge all matching events $e_{p^i}$ from $item.list[i]$.
Write merged string into output stream.
Increment all merged $p^i$s by 1.
Insert new entries from $t^i$s to $h$.
Update $item.\delta$ during insertion or removal from $h$.
```

be accessed only when it has reached the top of its process traces. Then the event will be continuously query for its $\delta(e)$ value until it is merged and emitted. In case there is no match for it in the current window, it will be immediately emitted since the distance is infinite, otherwise the number of accesses to an event is bound by the constant number of I/O function signature. Thus our amortized runtime is linear to E. In the worst case analysis, if only one event is emitted per iteration, the complexity however is $O(En)$, since each iteration costs $O(n)$. This worst case can be reduced further to $O(E)$ by carefully tracking whether all events at the head of each process trace is the same as well as the max distance whenever an event is added or removed from the hash table. However, in all of our experiments (up to 4096 trace files) the merging process was behaving linearly and there was no need to improve it further.

C. Code Generation

Once the merged trace is created by the merger, a code generator will generate a compilable Single-Program, Multiple Data (SPMD) code for it. The merged records which are called collectively by all the processors are easily generated in the I/O trace generator application. For the not-merged functions, there are number of ways one can take care of differentiating what each processors are doing:

1) **Using conditions:** The most straightforward solution to this problem is to use an `if - else` statement and put each of the ranks operations in their corresponding if clause. The problem with this approach is that code length is proportional to the number of processes, therefore for large-scale experiments the generated code will be very large.
2) **Using memory:** The second solution is to trade constant memory for code size. The way this works is that for every number or array which is different for different ranks, a new dimension is added corresponding to the rank of the MPI processes. This solution will decrease the size of generated code significantly, but still has some downsides such as requiring extra memory and a need to replicate this data on the memory of each MPI process.
3) **Identifying the relationship with MPI rank:** In most of the cases, there is a simple relationship between the offsets of the file a process is accessing and rank of that process. Therefore, the code generator can try to identify this pattern and find a relationship between these numbers. A math symbolic library can be used for this purpose. Currently, our code generator first checks for finding such a relationship to use. More specifically, it tries to check for this relationship for each of the dimensions of each of the different arguments of the functions that the merger has specified. In case it can not, it will fall back to the second solution and uses memory in order to differentiate different values for different processes.

Algorithm 3 getDistance method

```
Input: Trace event $e$ at $p^i$
Output: Distance value $\delta(e)$
Pseudo Code:
return $h(k(e)).\delta$
```

IV. Setup and Evaluation Results

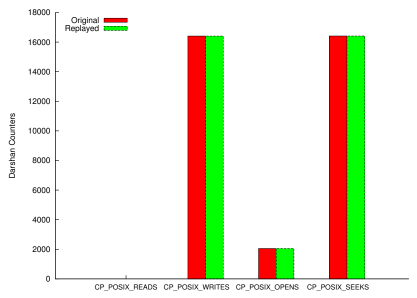
In this section we discuss the way that our framework performs for different applications at different scales. All the traces are gathered on the Stampede Dell cluster at Texas Advanced Computing Center (TACC). Stampede is a 10 PFLOPS computer consisting of more than 6400 nodes, each with 2 Intel Xeon E5 processors, with 16 cores per node.

Three different I/O kernels described below are used for this purpose. For each I/O kernel, we used our framework to generate an I/O trace generator and compared it to the original kernel. An advantage of these experiments is that it enables a fair comparison of I/O calls of the original I/O kernel with the generated I/O kernel using our framework. We ran all the three benchmarks on 2048 cores of Stampede each generating about 500 GB output file:

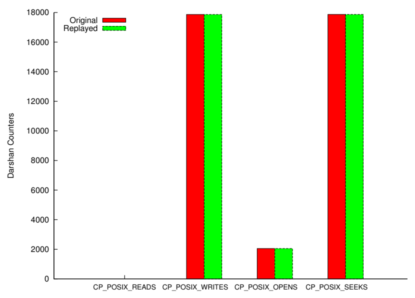
- **VPIC-IO:** The I/O kernel of a plasma physics application called VPIC, scalable parallel particle simulation application [6]. VPIC-IO uses H5Part [5] API developed at LBNL, wrapper library over HDF5 for particle data models. VPIC-IO is developed by extracting all the H5Part function calls of the VPIC code. The I/O motif of VPIC-IO is a 1D particle array of a given number of particles and each particle has eight variables and for each variable.
- **VORPAL-IO:** VORPAL-IO is an I/O kernel of a computational plasma framework application simulating the dynamics of electromagnetic systems named VORPAL developed by TechX [16]. This benchmark uses H5Block to write uniform chunks of 3D data per processor. H5Block is another library developed at LBNL used for block-structured data models.
- **GCRM-IO:** GCRM-IO is an I/O kernel simulating I/O for GCRM, a global atmospheric circulation model. It also uses H5Part to perform I/O operations. GCRM writes out a semi-structured geodesic mesh, with user-controllable resolution and subdomain resolution.

A. Correctness of the framework:

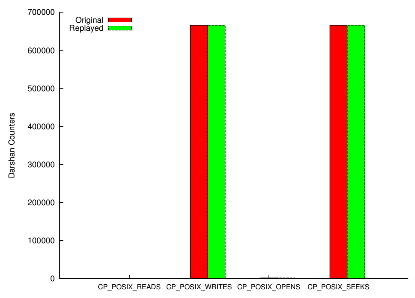
Figure 4(a) shows the comparison of some POSIX I/O counters of the original and generated VPIC-IO kernel. These



(a) VPIC-IO



(b) VORPAL-IO



(c) GCRM-IO

Fig. 4. Comparison of Darshan POSIX I/O counters of original application and the generated one by the Framework (a) VPIC-IO, (b) VORPAL-IO, (c) GCRM-IO

counters are derived using Darshan [7]. As it can be seen, values of these counters are exactly the same for the two kernels. The output files generated by the framework are also equal in size and have similar output for the `h5dump` tool, proving the correctness of the framework.

Figure 4(b) and 4(c) show the same comparison for the VORPAL-IO and GCRM-IO kernels respectively. In these case the framework has also been able to generate the same output file as the original VORPAL-IO and GCRM-IO applications both in terms of size and HDF5 file format.

B. Quality of the generated code:

The previous subsection showed the correctness of the framework for the three benchmarks; In this subsection we will look at the quality of the generated code represented by its size. Table I compares the size of of the original I/O benchmark source codes with the generated source code by our framework. As it can be seen VPIC-IO and GCRM-IO have generated code of size proportional to the original code (Original GCRM-IO code has more options causing the original code to be larger than the generated I/O benchmark). VORPAL-IO however has much larger generated source code. The reason for this is the complex relationship between the starting addresses of the 3D blocks assigned to the processes and their MPI ranks. The code generator was not able to find this relationship and had to fall back to using memory (solution #2) in order to generate correct code (since 2048 cores were used for these experiments, the code for initializing this array causes the large size of the source code). It is easy for the program developer to put this relationship in the generated code and reduce the size as the last column of Table I shows.

I/O Benchmark	Original Code	Generated Code	With user's help
VPIC-IO	8 KB	8 KB	8 KB
VORPAL-IO	12 KB	616 KB	36 KB
GCRM-IO	36 KB	12 KB	12 KB

TABLE I. Comparison of the source code size of Original and Generated I/O Benchmarks

V. Conclusions and Future Work

The use of I/O kernels is getting more popular in the HPC community. High-level I/O libraries with their higher productivity are also getting more popular as they show higher performance and simplify coding. There has been several efforts to automatically trace and replaying I/O operations of applications but they mostly focused on the lower-level layers of the I/O stack which is much more complex. In this work we have presented reasons, challenges and methods for creating I/O kernels at a high-level I/O library such as HDF5.

This framework consists of a recorder library to trace the higher-level I/O operations, a merger tool which merges traces recorded on each process, and a code generator generating the I/O kernel out of the merged I/O trace. Within each component, we have presented efficient algorithms and several optimizations which could be applied for any types of I/O libraries. We have also shown the applicability of this framework for three I/O benchmarks with very different I/O patterns.

There are different lines of work we plan to pursue as a follow-up to this work: We plan to improve the pattern matching capabilities of our framework, in order to be able to detect and compress more general control structures. While this "reverse engineering" of the control structure of the original program is not tractable, in general, we conjecture that the I/O of most scientific codes has simple control structures that can be detected by our methods.

Our work has focused on generating an I/O kernel for a fixed problem size and fixed number of processes. However, the same pattern matching techniques we use to compress traces can be used to detect dependencies on the number of processors or key input parameters. This will require multiple runs with different input size and different process counts.

Although we have shown that this framework works only for the HPC applications using HDF5 library, the tools and the operations in this work are all applicable to other high-level I/O libraries such as PnetCDF. This is also another future work that we consider, since adding such a capability will be very useful.

Last but not least, our grand vision for this research is to make use of this framework for different HPC applications in order to build a repository of I/O kernels that represent different HPC applications. This repository can be updated as the applications get updated and be used for different purposes such as I/O autotuning, storage systems evaluation, I/O performance analysis, etc. They can be used to characterize the impact of new technologies without requiring the developer scientists of the large-scale HPC applications to change their code.

VI. Acknowledgments

This work is supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract DE-AC02-05CH11231 and DE-AC02-06CH11307 and partly by NSF grant 0938064. It used resources of the Texas Advanced Computing Center. The authors would like to thank Huong Luu, Marianne Winslet, Bill Gropp, Prabhat, Suren Byna, and Quincey Koziol for their non-stop support and continuous helpful discussion.

References

[1] ADIOS 1.5 user's manual. http://users.nccs.gov/ pnorbert/adios-usersmanual-1.5.0.pdf.

[2] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 129–145, Berkeley, CA, USA, 2004. USENIX Association.

[3] B. Behzad, H. V. Dang, F. Hariri, W. Zhang, and M. Snir. Automatic generation of i/o kernels for hpc applications. In *Work-In-Progress at the 12th USENIX Symposium on File and Storage Technologies (FAST14)*, 2014.

[4] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir. Taming Parallel I/O Complexity with Auto-Tuning. In *Proceedings of 2013 International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2013)*, 2013.

[5] E. W. Bethel, J. M. Shalf, C. Siegerist, K. Stockinger, A. Adelmann, A. Gsell, B. Oswald, and T. Schietinger. Progress on H5Part: A Portable High Performance Parallel Data Interface for Electromagnetics Simulations. In *Proceedings of the 2007 IEEE Particle Accelerator Conference (PAC 07). 25-29 Jun 2007, Albuquerque, New Mexico. 22nd IEEE Particle Accelerator Conference, p.3396*, 2007.

[6] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas*, 15(5):7, 2008.

[7] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Understanding and improving computational science storage access through continuous characterization. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, 2011.

[8] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snirt, B. Traversat, and P. Wong. Overview of the mpi-io parallel i/o interface. In *Input/Output in Parallel and Distributed Computer Systems*, pages 127–146. Springer, 1996.

[9] M. Folk, A. Cheng, and K. Yates. HDF5: A file format and I/O library for high performance computing applications. In *Proceedings of Supercomputing*, volume 99, 1999.

[10] G. R. Ganger. Generating representative synthetic workloads: An unsolved problem. In *in Proceedings of the Computer Measurement Group (CMG) Conference*, pages 1263–1269, 1995.

[11] M. Howison, Q. Koziol, D. Knaak, J. Mainzer, and J. Shalf. Tuning HDF5 for Lustre File Systems. In *Proceedings of 2010 Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS10)*, Heraklion, Crete, Greece, Sept. 2010. LBNL-4803E.

[12] 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 *SC Companion*, pages 18–23. IEEE Computer Society, 2012.

[13] J. Logan, S. Klasky, J. Lofstead, H. Abbasi, S. Ethier, R. Grout, S.-H. Ku, Q. Liu, X. Ma, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. Skel: Generative software for producing skeletal i/o applications. In *Proceedings of the 2011 IEEE Seventh International Conference on e-Science Workshops*, ESCIENCEW '11, pages 191–198, Washington, DC, USA, 2011. IEEE Computer Society.

[14] H. Luu, B. Behzad, R. Aydt, and M. Winslett. A multi-level approach for understanding i/o activity in hpc applications. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–5, 2013.

[15] M. P. Mesnier, M. Wachs, R. R. Sambasivan, J. Lpez, J. Hendricks, G. R. Ganger, and D. O'Hallaron. //TRACE: Parallel Trace Replay with Approximate Causal Events. In *FAST*, pages 153–167. USENIX, 2007.

[16] C. Nieter and J. R. Cary. VORPAL: a versatile plasma simulation code. *Journal of Computational Physics*, 196:448–472, 2004.

[17] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, volume 2003, 2003.

[18] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '99, pages 182–, Washington, DC, USA, 1999. IEEE Computer Society.

[19] S. A. Wright, S. D. Hammond, S. J. Pennycook, R. F. Bird, J. A. Herdman, I. Miller, A. Vadgama, A. Bhalerao, and S. A. Jarvis. Parallel File System Analysis Through Application I/O Tracing. *Comput. J.*, 56(2):141–155, 2013.

[20] X. Wu, K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth. Probabilistic Communication and I/O Tracing with Deterministic Replay at Scale. In *ICPP*, pages 196–205. IEEE, 2011.