Feign: In-Silico Laboratory for Researching I/O Strategies
Using the Flexible Event Imitation Engine (Feign) to alter application I/O

Jakob Lüttgau
Universität Hamburg
Bundesstr. 45a
20146 Hamburg
Email: jakob@whatz.eu

Julian M. Kunkel
DKRZ
Bundesstr. 45a
20146 Hamburg
Email: kunkel@dkrz.de

***Abstract*—Evaluating I/O performance of an application across different systems is a daunting task because it requires preparation of the software dependencies and required input data. *Feign* aims to be an extensible trace replay solution for parallel applications that supports arbitrary software and library layers. The tool abstracts and streamlines the replay process while allowing plug-ins to provide, manipulate and interpret trace data. Therewith, the application's behavior can be evaluated without potentially proprietary or confidential software and input data.**

Even more interesting is the potential of Feign as a virtual laboratory for I/O research: by manipulating trace data, experiments can be conducted; for example, it becomes possible to evaluate the benefit of optimization strategies. Since a plug-in could determine "future" activities, this enables us to develop optimal strategies as baselines for any run-time heuristics, but also eases testing of a developed strategy on many applications without modifying them.

The paper proposes and evaluates a workflow to automatically apply optimization candidates to application traces and approximate potential performance gains. By using Feign's reporting facilities, an automatic optimization engine can then independently conduct experiments by feeding traces and strategies to compare the results.

I. Introduction

Today most high-performance computers (HPC) are cluster installations [14]. Clusters introduce complexity on many levels because of the variety of hardware and software technologies they utilize. Since applications pose different demands on the system, performance prediction is anything but trivial. A classification scheme for scientific applications is offered by the thirteen dwarfs[4]; they also serve as templates for application-specific benchmarks. Storage and network technologies are bottlenecks in modern HPC systems; in general, data can be produced much faster than permanently stored.

New technology is arising all the time, as do problems with bleeding edge hardware and software. Evaluating these, or comparing new technology with already deployed systems is often non-trivial. Also, to validate research in I/O middleware, it is important to check its impact on many applications. The system's complexity requires in-dept knowledge, the same is also true for preparing the required environment for an application. Complicated dependencies can make deploying an application on another system unfeasible, especially when human resources are already scarce. Finally, data and/or programs might be confidential so that giving away copies is not an option. This makes seeking support within the open source community or even from vendors more complicated.

Comparing systems is traditionally done using benchmarks. But synthetic benchmarks such as the LINPACK for HPC systems are not well suited to predict the behavior of applications. Developing application-specific benchmarks is very time consuming. One approach to automate and simplify application specific benchmarking, stress-testing and debugging is to make use of application traces. This overcomes many of the problems mentioned above.

Traces, being available in binary or text formats, are very portable in comparison to applications. Also, once compiled, a generic trace player could mimic many applications with little effort. Traces reliably capture the application characteristics. In fact, some of the benchmarking tools in Section II already use application traces to stress the system. It is also easy to remove confidential information, as in the particular case of I/O, only the access pattern is relevant but not the actual data.

Traces are deterministic and invariant by themselves which is a benefit in many situations, but at the same time also introduces constraints. For example, traces are to some extent bound to the environment where they have been obtained – e.g., already existing input files. Thus, replaying the trace on another system may not be so meaningful after all.

Another issue is the evaluation of code-invasive optimization strategies; To test a certain strategy such as application guided pre-fetching, the code must be changed without knowing the benefit in advance. It is prohibitively time consuming to test the all s strategies with all a applications, as it would require $(a \cdot s)$ implementations.

The paper illustrates how these limitations can be relaxed and, more importantly, how to use trace data to conduct I/O related experiments without needing to change the application code, developing plug-ins for *Feign* that look for patterns in the trace sequence and apply optimizations automatically where appropriate. Therewith, strategies can be developed orthogonally to application code and after approximating the benefit, fitting optimizations can be integrated into the applications (or, preferably, the middleware).

This paper is structured as follows: related work will be presented in Section II, covering tools to obtain traces as well as existing benchmarking and replaying solutions. Section III will introduce *Feign*, the *flexible event imitation engine*. Section IV presents applications for replay software beyond stoically reproducing system activity; in particular, we suggest applying optimization strategies to trace data, similar to conducting experiments in a virtual laboratory. Examples of this approach are given and evaluated in Section V, before we conclude with an outlook on future work in Section VI.

II. Related Work

Related work can be classified into approaches for tracing, benchmarking and specific replaying tools.

Monitoring and analysis of system state and performance data are important for optimizing HPC systems; tools which record application and I/O behavior include Vampir [9], ITAC [7], Darshan [5] and SIOX [10]. Looking beyond HPC I/O, trace tools such as Wireshark [6] for network analysis can be considered. Many domain-specific workload catchers are available, e.g., NFS workloads [3], but typically they do not collect all the information needed for replay. It is also important to realize that replay authenticity directly depends on the granularity and timing precision of the trace environment.

Communities may extract specific benchmarks directly from an application. For example, the (outdated) FLASH I/O kernel [17] performs a typical HDF5 I/O pattern as created by the FLASH astrophysics community code. The RAPS[1] initiative started at the end of the 20th century [13] tries to extract benchmarks by reducing the code base of an application to its significant part. However, for many applications, I/O kernels are not available.

There are also tools and environments which reduce the burden of creating a benchmark or application kernel. Parabench [12] is a programmable parallel benchmark. Benchmarks can be programmed and stored in a special script language. It also permits us to set up MPI groups and to generate reports from the measurements. Several tools allow replay of a single process's access pattern, for example, IOZone[2] or ioreplay [1]; IOZone replays telemetry data (offset, size, delay) from a file. However, only few are available in the domain of parallel computing.

DIOS [8] is a parallel file system benchmark developed during a PhD thesis at the TU Dresden. Instructions are read from XML input, which also allow to demand parameter randomization. POSIX replay is realized by converting VampirTrace [2] files into XML. It supports MPI parallel tasks as well as MPI-I/O and POSIX I/O. The tool has not officially been released yet. The skel tool [11] allows for creating ”skeleton applications” in an automatic way through multiple processing steps based on ADIOS XML configuration files.

Unfortunately, the tools that are publicly available are hard to extend. But monolithic solutions are impractical when it comes to meet the diverse demands of parallel systems.

A. *SIOX*

Since traces created by the SIOX project are used with *Feign* for evaluation (cf. Section V), the following subsection will describe SIOX in more detail. SIOX [10] is a recent project combining online monitoring with offline learning. It collects event-based information about the I/O activities taking place at the instrumented components of a system, and samples statistics about the resource usage of the nodes where the SIOX daemons are running. Monitoring data flows from the instrumented component into the daemon process, and from there into one of SIOX's transaction servers [15]. The recorded information can be analyzed offline to update a knowledge base holding optimized parameter suggestions for common or critical situations. During online operations, these parameters may be queried and used as predefined responses whenever such a situation occurs. Several optimization plug-ins are currently under development and assessment. The choice of responses to each situation is diverse, ranging from adjusting the monitoring level in the presence of anomalies, to automatically enforcing optimization techniques to achieve better performance [16].

[1]Real Applications on Parallel Systems

[2]http://www.iozone.org/

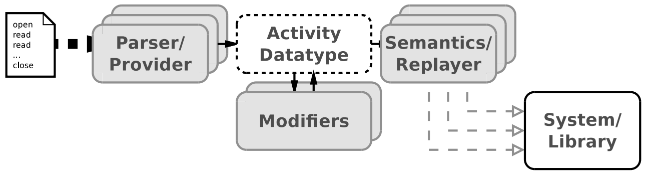

Fig. 1. Separating the concern of reading trace files from interpreting, replaying and modifying by providing an abstract datatype.

SIOX was designed with modularity in mind: Upon startup of either a process, component, or daemon, a configuration file is read containing the desired modules and plug-ins that are to be used. Modules may offer additional functionality for analysis or optimizations. A holistic approach to tracing is offered by SIOX which makes it attractive as a source for application traces from a replaying perspective. Users of SIOX can customize the instrumentation and thus derive arbitrary granularity. The only limit here is system performance and latency and distortions introduced during tracing. It turns out that a replay engine can integrate nicely into an optimization framework (cf. Section IV).

III. Flexible Event Imitation

While replaying traces seems to be a straightforward task which can be fully automated, in practice, a lot of preprocessing and manual work is still left to the user. A first set of problems stems from the variety of different trace formats that are available as well as the need to meet certain pre-requisites in the environment. Furthermore, managing requirements such as pre-creation of files and directories that are read may be necessary depending on the layer. Luckily, many of these problems can be automated, but so far, little effort is invested to consolidate different strategies concerning replay.

Feign establishes a processing pipeline to allow for meaningful playback. The replay process is split into multiple phases which plug-ins can hook into. By separating the interpretation of an activity stream from parsing the trace files, an interpreter for one layer can replay activities regardless of the source, provided all necessary information is available in the trace format. By offering operations to modify the trace, it becomes possible to account for system details and to add or retract system-specific optimizations as well as variations and

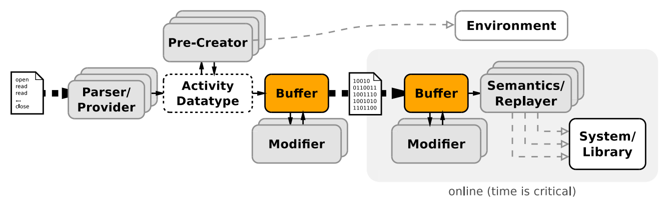

Fig. 2. The activity pipeline populated with several plug-ins.

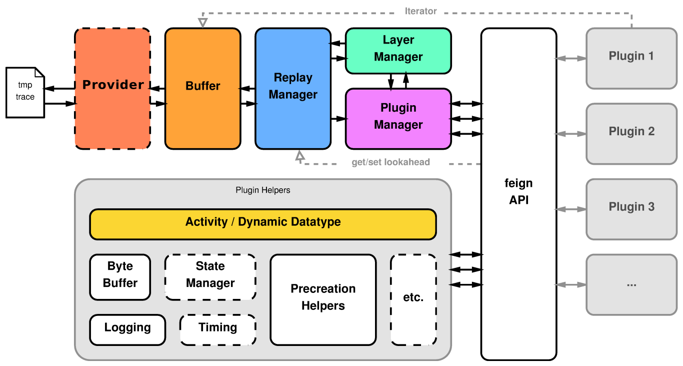

Fig. 3. The *Feign*-API and how plug-ins can interact with the core components, e.g., to increase the size of the lookahead window.

randomness. An abstract view of Feign's modular approach is shown in Figure 1.

Many traces are not directly replayable because they make requirements on the environment, e.g., replaying a file which is read requires the file to exist and be of a certain size. These requirements are likely not to be met when moving between systems. Similar problems arise when replaying MPI, while trivial cases of `MPI_Send()` and `MPI_Recv()` work out of the box with Feign. Calls that use previously registered MPI datatypes or communication groups are only replayable in a meaningful way when the trace environment records calls to the functions responsible for their creation and registration.

To replay such traces, some additional effort is required. As this is pretty common, *Feign*'s replay pipeline provides means to establish replayability by incorporating a pre-creation phase. This phase can also be used to clean the trace of uninteresting activities. Moreover, since modifier plug-ins may need some CPU time to alter or inject activities, it is possible to perform a first run in which these time-consuming tasks are performed and then store a modified trace file which is used during the replay in a second phase. This way, many trace files regain replayability and the actual replay is more accurate as unnecessary distortions from polluted trace files can be minimized. Figure 2 gives a graphical overview of the overall replay pipeline.

Figure 3 provides an overview of Feign's APIs. A *provider* allows for reading a trace or creating activities on the fly to mimic arbitrary application configurations. Several activities are held in the buffer which offers a convenient look-ahead for the plug-ins. E.g., a plug-in can inject activities if a specific condition is met. The replay manager controls fetching new activities and invokes a fitting *replayer* for each particular layer (POSIX, MPI, ...). Feign allows plug-ins to hook into different phases of the replay which eases creation of new tools.

```
#include <feign.h>

// provide some meta information
Plugin plugin = {
   .name = "Example-Replayer",
   .version = "1.2.3",
   .intents = FEIGN_REPLAYER,
};

int init(int argc, char *argv[])
   feign_register_plugin(&plugin);
   return 0;
}

// expected because of FEIGN_REPLAYER
Activity * replay(Activity * activity) {
   // do something and consume activity
   return NULL;
}
```

Fig. 4. A minimalistic example for a replay plug-in. In `init()` the plug-in announces that it intends to (just) replay activities. Feign then locates the symbol for `replay()` and this callback is registered in the processing pipeline (see Figure 2).

Several helper tools are available (or planned) to simplify the development of new plug-ins. *Pre-creation helpers*, for example, simplify setting up the initial environment before measuring the performance.

Creating a plug-in in C/C++ is straightforward – developers need to include the `feign.h` header file and define a function called `init()` where the plug-in can announce any further intents. For each intent, Feign expects a callback function which is registered into the activity pipeline (see Figure 2). A very simple example is shown in Figure 4. To use the plug-in, the source needs to be compiled and linked into a shared object. The Feign executable allows the user to specify different plug-ins to load using the `--plugin <path-to-shared-object>` argument.

To ease creation of basic provider and replay plug-ins, we work on a tool which supports automatic plug-in generation from annotated header files. These annotations are very similar to those used in the SIOX project and we envision an annotation format which supports both, creation of monitoring plug-ins for SIOX and creation of different Feign plug-ins for reading, executing and manipulating previously recorded activities. Instead of creating plug-ins, the same tool could also change the output flavor directly creating C code that represents the access pattern. Currently, however, this is only partly achieved and the subject of further development.

IV. Virtual Laboratory

Testing an optimization strategy on an application may require intrusive code changes, and a strategy needs to be implemented in each application that may benefit from it. This is not only a very tedious task, but may cause situations in which very good strategies are not implemented because nobody could predict their benefit. Moreover, sometimes it is interesting to derive optimal strategies that are baselines for any online heuristics such as caching. Normally, an optimal strategy would require an oracle which predicts the future

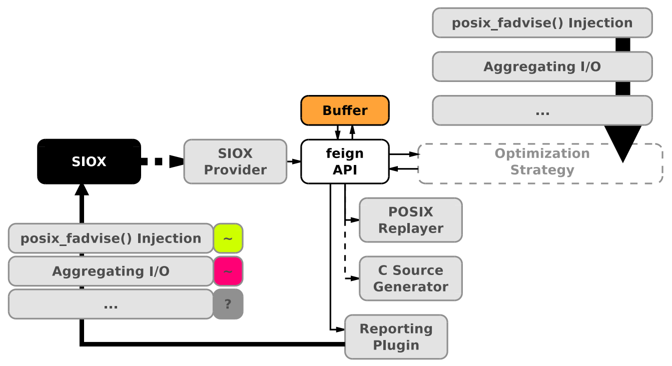

Fig. 5. Virtual laboratory: A third-party application uses Feign to explore parameters and strategies.

application behavior accurately. Traces, however, contain all the future activities and, thus, can fill this role.

Additionally, universal as well as context-dependent optimizations can be isolated and used in automatic optimization engines to evaluate their benefit for all applications. *Feign* allows users to create plug-ins that apply optimization strategies to trace data. These optimizations can be tailored or parameterized to take system details into account. Currently, we have implemented a pre-fetching strategy using `posix_fadvise(POSIX_FADV_WILLNEED)` and a module which aggregates concurrent I/O operations into bigger calls.

In many situations, however, there is a lack sufficient models to decide which optimizations to apply. In such situations, it is common to turn to experiments. But the number of parameters usually involved prohibits brute force search, and in spite of the latest advances in machine learning, intelligent search space exploration is still in its infancy.

Approaches like reinforcement learning become easy to use when there is a playground to experiment and try different strategies, thus creating a feedback loop allowing the system to adjust and even actively learn to some extent for itself.

An optimization system can identify problems for example by considering peak or average performance data. A detected anomaly potentially offers optimization opportunities. For instance, if SIOX detects such an anomaly, it might have an optimization at hand and apply it, but often the decision component within could not decide what to do. In this case, the framework could trace application behavior, partly or in total, for later testing with Feign. This workflow is illustrated in Figure 5.

V. Evaluation

To demonstrate the viability of the concept, we explored two basic optimization strategies. These examples may seem simplistic, but keep in mind that the I/O calls may be distributed across the application logic as well as across many files, resulting in high implementation effort even though the strategy is simple. For the following examples, all traces are obtained using SIOX. To gain replayability, the particular settings made use of a ”change I/O root”-plug-in, that rewrites all file paths to point to a user defined directory.

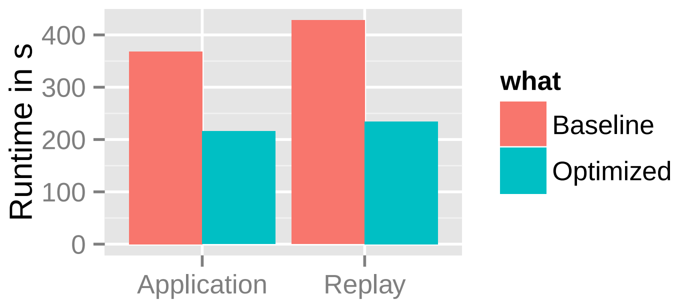

Fig. 6. Application and replay runs compared with and without optimizations. In the case of replay the `posix_fadvise(POSIX_FADV_WILLNEED)` is automatically injected using a *Feign* plug-in looking for a `lseek();read();` pattern.

A. *Pre-fetching of Data*

There are many strategies for pre-fetching. Application-aware pre-fetching promises to achieve best performance, but requires the programmer to modify the application: `posix_fadvise(POSIX_FADV_WILLNEED)` allows the programmer to announce regions of the file that will be needed in the near future and thus are candidates for pre-fetching. Linux may pre-fetch the data and thus reduce wait times for the program, resulting in a notable speedup. Thanks to Feign, potentially interesting strategies can be evaluated prior to coding.

A naive strategy is to look two activities ahead: if it is an `lseek()` followed by a `read()`, an fadvise call is injected with the particular offset and the size specified by the read call. Computation time is not taken into account explicitly; if there is sufficient compute time between the current activity and the `read()`, pre-fetching can completely hide the access of slow I/O.

This simple approach just pre-fetches the next operation and assumes free memory suffices. A more sophisticated strategy would not only look at the next two activities, but scan more. Based on the time between them, the advise needs to be injected a few activities before the actual `read()`. Moreover, by injecting multiple advises, the OS and disk scheduler can optimize the access order better.

To assess this strategy, a small benchmark is created which loops over 10 GiB of data stored on a local disk. On each iteration, the benchmark simulates compute time by sleeping for a while, then it seeks 1000 KiB forward and reads a 1 KiB chunk – the whole area covered by the seek and one access is defined as the stride size. The sleep time is 10 ms to make read-ahead possible. The benchmark is executed several times; between runs, the page cache is cleared using `echo 3 > /proc/sys/vm/drop_caches`.

In order to validate the results, the calls to `posix_fadvise(POSIX_FADV_WILLNEED)` have been also embedded into the benchmark's source code, yielding the best performance. Figure 6 shows the runtime of the application with and without optimization, and the replay. As can be seen, appropriate pre-fetching can speed up this particular application by almost a factor of two (this is rather obvious as I/O time can be completely hidden). With our first prototype, there is a noticeable overhead in the replay baseline runs; the reason for the effect is not investigated in detail, yet. However, validation runs, in which just the actual I/O and sleep calls have been removed, showed only a small overhead of around 2.5%.

Scalability is an important requirement for any workload replay engine in HPC. Using pre-processing, each rank can be provided with a trace stripped of all unrelated activity and modifications applied. If the nodes come with local storage, I/O can be further reduced by moving the optimized trace files from the shared storage to node local storage.

B. Aggregating I/O

Coalescing is a common optimization strategy where adjacent data accesses can be merged into one larger access to reduce the number of system calls. While the basic strategy is already employed in kernel space, applications that issue many small operations still suffer from the costs involved in a system call which can be easily avoided by accessing larger chunks of data. Moreover, coalescing of read operations is often not possible in kernel space, because it is impossible to predict the next position of future reads.

The developed coalescing plug-in merges adjacent reads or writes until a certain buffer size is reached and then issues one call for all of them instead. In this experiment, the access pattern of the FLASH HDF I/O kernel is traced for 8 processes. We pick the trace of one process and replay it using different buffer sizes. The observed run-time in shown in Figure 7. With buffers in the range of several 100 KiB, a performance improvement of 14 percent can be achieved. This is mainly due to the fact that out of 782 issued writes, 640 merely append 4 or 8 byte. The trace for this experiment does include the actual (input) data, by knowing the bytes read and written, the trace engine can generate dummy data in a pre-creation step: A file can be initialized with a size according to the last byte accessed by any `read()` in the trace.

If read-only access to high-level APIs such as HDF5 should be recorded and replayed, the pre-creation of the exact file layout is impossible without additional knowledge: a trace may just document the access of one out of many scientific variables stored in the logical file, or the file layout may depend on settings given during the initial creation of the file. Therefore, handling this kind of workload requires the trace environment to record the actual settings for the pre-creation. Alternatively, a Feign plug-in may use a default layout or allow the user to define the layout.

C. Replay of Parallel Applications: MPIOM

To demonstrate the benefit of Feign, the Max-Planck-Institute Global Ocean Model (MPIOM) was run on 10 nodes using SIOX to obtain the trace. In this case, all activity was recorded into a single trace file and the trace is then replayed on another test system using Feign. To replay only the activity of certain processes, a simple filter plug-in that white-listed only a specific rank was used, thus dropping any unwanted activity. For MPIOM, this is desirable because only a few nodes (in our case only one) act as application-specific I/O servers, which allows us to investigate and experiment with the I/O pattern on smaller systems. The processing pipeline also includes the plug-in for providing activities from the SIOX trace and another plug-in to replay POSIX activities.

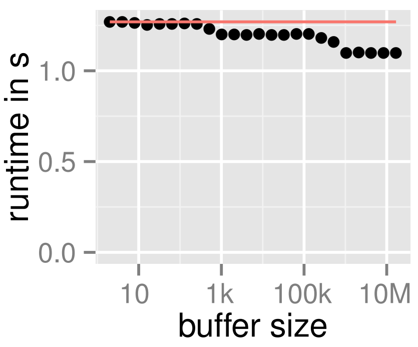

Fig. 7. Evaluating coalescing with different buffer sizes for a FLASH I/O replay. The red line is the run-time without aggregation.

The optimization plug-ins introduced in this paper did not result in a runtime reduction for the trace, as the workload is dominated by larger writes. However, the replay was able to recreate the workload of the application on the filesystem (created files, read/write access patterns) for the annotated functions. Prior to actual replay, a precreator plug-in checks the target directory, creating required input files of sufficient size.

VI. Summary & Outlook

Our results encourage to continue following the proposed approach. Extracting I/O benchmarks from applications can be automatized by tracing the behavior and by executing the trace in a replay engine. This removes external dependencies of the original application and eases communication with vendors by extracting relevant code pieces for later investigation. The Feign tool allows not only to replay traces but also to alter them on-line or off-line. Moreover, Feign provides everything necessary for conducting I/O research in a virtual laboratory. Various optimization strategies can be integrated; it is possible to inject additional system hints or rewrite sequences of operations. Implementing plug-ins is reasonable easy, and, they can be applied to other traces, by simply switching them on or off. Two basic strategies – coalescing I/O accesses and pre-fetching – have been integrated and evaluated on simple test-cases. Since optimization strategy and application trace can be regarded as orthogonal in our approach, the effect of intrusive code modifications can be evaluated before actually changing the application.

Currently, we are unifying the annotation system between SIOX and Feign to automatically create tracing and replay plug-ins. Layer support for a subset of POSIX calls is already achieved using automatic generation from annotated header files. Better POSIX and MPI support as well as generation for filtering and mutation plug-ins are next on the roadmap. We also experiment with alternative optimization schemes for several traces of climate applications, we will harden the software core and optimize it. For the future, we envision a platform for exchanging traces and optimization strategies in order to raise portable HPC optimization to a new level.

ACKNOWLEDGMENTS

We thank Venka Michaela Zimmer for her review.

REFERENCES

[1] ioreplay. http://code.google.com/p/ioapps/wiki/ioreplay, 2014. [Online; accessed 2014-03-01].

[2] Vampir Trace. http://www.vampir.eu/, 2014. [Online; accessed 2014-03-14].

[3] E. Anderson. Capture, Conversion, and Analysis of an Intense NFS Workload. pages 139–152, 2009.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, and C. Sciences. The Landscape of Parallel Computing Research: A View from Berkeley. 2006.

[5] P. H. Carns, R. Latham, R. B. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale I/O workloads. In *Proceedings of the First Workshop on Interfaces and Abstractions for Scientific Data Storage*, New Orleans, LA, USA, Sept. 2009.

[6] G. C. et al. Wireshark, network protocol analyzer. https://www.wireshark.org, 2014.

[7] Intel. ITAC, Intel Trace Analyzer and Collector. https://software.intel.com/en-us/intel-trace-analyzer, 2014.

[8] M. Kluge. Comparison and End-to-End Performance Analysis of Parallel File Systems. (September), 2011.

[9] 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, Proceedings of the 2nd International Workshop on Parallel Tools*, pages 139–155. Springer, 2008.

[10] J. Kunkel, M. Zimmer, N. Hübbe, A. Aguilera, H. Mickler, X. Wang, A. Chut, T. Bönisch, J. Lüttgau, R. Michel, and J. Weging. The SIOX Architecture – Coupling Automatic Monitoring and Optimization of Parallel I/O. In *Supercomputing*, number 8488 in Lecture Notes in Computer Science, pages 245–260, Berlin, Heidelberg, 06 2014. Springer.

[11] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, M. Parashar, N. Samatova, K. Schwan, A. Shoshani, M. Wolf, K. Wu, and W. Yu. Hello ADIOS: The Challenges and Lessons of Developing Leadership Class I/O Frameworks. 2013.

[12] J. May and L. Livermore. Pianola: A script-based I/O benchmark. *International Symposium on Automated Analysis-Driven Debugging*, pages 69–76, 2005.

[13] G. Mozdzynski. RAPS. Presentation: https://www.irisa.fr/orap/Forums/Forum20/Presentations/GeorgeMozdzynski.pdf, 2006.

[14] Top500. Top500. http://www.top500.org/, 2014. [Online; accessed 2014-03-06].

[15] M. C. Wiedemann, J. M. Kunkel, M. Zimmer, T. Ludwig, M. Resch, T. Bönisch, X. Wang, A. Chut, A. Aguilera, W. E. Nagel, M. Kluge, and H. Mickler. Towards I/O Analysis of HPC Systems and a Generic Architecture to Collect Access Patterns. *Computer Science - Research and Development*, 1:1–11, 2012.

[16] M. Zimmer, J. Kunkel, and T. Ludwig. Towards Self-optimization in HPC I/O. In *Supercomputing*, number 7905 in Lecture Notes in Computer Science, pages 422–434, Berlin, Heidelberg, 06 2013. Springer.

[17] M. Zingale. FLASH I/O Benchmark Routine Parallel HDF 5. http://www.ucolick.org/~zingale/flash_benchmark_io/, 2001.