The SIOX Architecture – Coupling Automatic Monitoring and Optimization of Parallel I/O

Julian Kunkel[1], Michaela Zimmer[1], Nathanael Hübbe[1], Alvaro Aguilera[2], Holger Mickler[2], Xuan Wang[3], Andriy Chut[3], Thomas Bönisch[3], Jakob Lüttgau[1], Roman Michel[1], and Johann Weging[1]

[1] University of Hamburg
[2] ZIH Dresden
[3] HLRS Stuttgart *

Abstract. Performance analysis and optimization of high-performance I/O systems is a daunting task. Mainly, this is due to the overwhelmingly complex interplay of the involved hardware and software layers. The Scalable I/O for Extreme Performance (SIOX) project provides a versatile environment for monitoring I/O activities and learning from this information. The goal of SIOX is to automatically suggest and apply performance optimizations, and to assist in locating and diagnosing performance problems.
In this paper, we present the current status of SIOX. Our modular architecture covers instrumentation of POSIX, MPI and other high-level I/O libraries; the monitoring data is recorded asynchronously into a global database, and recorded traces can be visualized. Furthermore, we offer a set of primitive plug-ins with additional features to demonstrate the flexibility of our architecture: A surveyor plug-in to keep track of the observed spatial access patterns; an fadvise plug-in for injecting hints to achieve read-ahead for strided access patterns; and an optimizer plug-in which monitors the performance achieved with different MPI-IO hints, automatically supplying the best known hint-set when no hints were explicitly set. The presentation of the technical status is accompanied by a demonstration of some of these features on our 20 node cluster. In additional experiments, we analyze the overhead for concurrent access, for MPI-IO's 4-levels of access, and for an instrumented climate application. While our prototype is not yet full-featured, it demonstrates the potential and feasibility of our approach.

Keywords: Parallel I/O, Machine Learning, Performance Optimization

1 Introduction

I/O systems for high-performance computing (HPC) have grown horizontally to hundreds of servers and include complex tiers of 10,000 HDDs and SSDs. On

* We want to express our gratitude to the „Deutsches Zentrum für Luft- und Raumfahrt e.V." as responsible project agency and to the „Bundesministerium für Bildung und Forschung" for the financial support under grant `01 IH 11008 A-C`.

the client side, complexity of high-level software layers increases. This leads to a non-trivial interplay between hardware and software layers, and diagnosing it has become a task to challenge even experts. Parameterizing the layers for optimum performance requires intimate knowledge of every hardware and software component, including existing optimization parameters and strategies.

The SIOX Project was initiated to shed light on the interactions, and to offer automatic support for optimizing the HPC-I/O stack. Continually monitoring performance and overhead, an I/O system instrumented with SIOX will autonomously detect problems, inferring advantageous settings such as MPI hints, stripe sizes, and possible interactions between them. In this paper we will present first results obtained with the SIOX prototype.

The contributions of this paper are: 1) a description of our modular architecture for monitoring, analysis and optimization. 2) analysis of the overhead for synthetic benchmarks and a climate model. 3) use-cases demonstrating the benefit of automatic optimization.

This paper is structured as follows: Section 2 sketches the state of the art in I/O performance analysis. The modular architecture and implementation of SIOX is introduced in Section 3. Section 4 describes tools to analyze and visualize instrumented applications. We evaluate the overhead of our prototype in Section 5, and demonstrate the potential of this approach in several scenarios. Finally, ongoing and future work is discussed in Section 6 and conclude our experience with the SIOX prototype in Section 7.

2 Related Work

Efforts to monitor I/O behavior are legion, the latest widely-used exponent being Darshan [1], a lightweight tool to gather statistics on several levels of the I/O stack, primarily MPI and POSIX.

Early approaches to true system self-management relied on the direct classification of system state or behavior to automatically diagnose problems or even enact optimization policies. The work of Madhyastha and Reed [2] compares classifications of I/O access patterns, from which higher level application I/O patterns are inferred and looked up in a table to determine the file system policy to set for the next accesses. The table, however, has to be supplied by an administrator implementing his own heuristics.

Later approaches are marked by schemes to persist their results in order to benefit from past diagnostic efforts, possibly even applying known repair actions to recognized problems. `Magpie`, a system by Barham et al. [3], traces events under Microsoft Windows, merging them according to predefined schemata specifying event relationships. Their causal chains are reconstructed, attributed to external requests and clustered into models for the various types of workload observed. Deviations will point to anomalies deserving human attention.

Yuan et al. [4] combine system state and system behavior to identify the root causes of recurring problems. Tracing the system calls generated under Windows XP, they use support vector machines to classify the event sequences.

A presumptive root cause is identified, leaving the sequence – if flagged by a human as accurately diagnosed – available as eventual new training case for the classifier. The root cause description may include repair instructions, which in some cases can be applied automatically.

Of the systems focusing on system metrics, `Cluebox` by Sandeep et al. [5] analyses logs for anomalies, pointing out the system counters most likely involved in the problem. Expected latencies can be predicted for new loads, not only detecting anomalies but also which counters most significantly deviate from par. No direct tracing or causal inference are needed, but once again, only hints for administrators are produced. In `Fa` (Duan et al. [6]), a system's base state is defined by service level objectives; compliance constitutes health, violation failure. A robust data base of failure signatures is constructed from periodically sampled system metrics.

Behzad et al. [7] offer a framework that uses genetic algorithms to auto-tune select parameters of an HDF5/MPI/Lustre stack; but its monolithic view of the system disregards the relations between the layers as well as the users' individual requirements, setting optimizations once per application run.

Valuable capabilities of existing approaches are combined in the SIOX infrastructure and extended by unique features: 1) SIOX covers parallel I/O on client and server level as well as intermediate levels, 2) it aims to be applicable at all granularities and portable across middle-ware and file systems, 3) SIOX does not require MPI and can be applied to POSIX applications easily 4) it unites user-level and system-level monitoring supporting both views, 5) it applies machine-learning strategies to learn optimizations on-line and off-line and apply them – ultimately without human intervention, 6) SIOX utilizes a system model to estimate performance, 7) it restricts monitoring to relevant anomalies, 8) finally, SIOX is extremely modular and its capabilities can be configured for many different use-cases.

3 The Modular Architecture of SIOX

Under SIOX, a system will collect information on I/O activities at all instrumented levels, as well as relevant hardware information and metrics about node utilization. The high-level architecture of SIOX has been introduced in [8]: SIOX combines on-line monitoring with off-line learning; monitoring data is first transferred into a transaction system, and then imported into a data warehouse for long-term analysis. The recorded information will be analyzed off-line to create and update a knowledge base holding optimized parameter suggestions for common or critical situations. During on-line operations, these parameters may be queried and used as predefined responses whenever such a situation occurs. The choice of responses to every situation is diverse, ranging from intelligent monitoring in the presence of anomalies, via alerting users and administrators with detailed reports, to automatically taking action to extract the best possible performance from the system. In Zimmer et.al. [9] we discussed the knowledge path in more detail, and sketched several modules for anomaly detection. This paper

extends our previous theoretical articles by describing our existing prototype, and presenting first results.

SIOX is written in C++, and heavily relies on the flexible concept of modules: Upon startup of either a process, a component, or the daemon, a configuration is read which describes the modules that must be loaded. Several modules offer an interface for further plug-ins which, for example, may offer specialized detection of anomalies, and may trigger actions based on the observed activities and system state. Before we outline the currently existing modules and plug-ins, the instrumentation of an application is described.

3.1 Low-level API

An instrumented application is linked against at least one instrumented software layer, and against the low-level C-API. Upon startup of the SIOX low-level library, a configuration file is read, and globally required modules are loaded. Whenever a logical component – such as the MPI layer – registers, a component-specific section in the configuration is read, and the layer-specific modules described there are loaded. The following interfaces are directly required by the low-level library:

- *Ontology*: The ontology module provides access to a persistent representation of activity attributes such as function parameters.
- *SystemInformationGlobalIDManager*: Provides a means to map existing physical hardware (nodes, devices), and software components (layer and existing activity types) to a global ID, and vice versa.
- *AssociationMapper*: While the Ontology and SystemInformation are rarely updated, runtime information of an application changes with each execution. Therefore, this data is held in a separate store with an efficient update mechanism, both provided by the AssociationMapper.
- *ActivityMultiplexer*: Once an activity is completed, it is given to the multiplexer, which forwards it to all connected MultiplexerPlugins.
  Each of these analyzes the data of the incoming activities to fulfill higher-level duties. The multiplexer offers both, a synchronous and an asynchronous data path. The latter allows for more performance-intense analysis, but may incur loss of activities when the system is overloaded.
- *Optimizer*: The optimizer provides a lightweight interface to query the best known value for a tunable attribute. Internally, a value is provided by a plug-in; each plug-in may support a set of attributes. The value may depend on the system status, runtime information about the process, or sequence of activities observed.
- *Reporter*: Modules can collect some information about their operation, such as overhead, and amount of processed data. Upon termination of an application, this data is handed over to so-called reporter modules. A reporter may process, store, or output these statistics, according to its configuration.

Additionally, the library uses some helper classes to build activities, to load and manage modules, and to monitor the internal overhead.

3.2 Instrumentation

When tracing foreign code, keeping the instrumentation up to date can become a maintenance problem. To keep manual work to a minimum, SIOX automatically generates instrumentations for layers from annotated library headers. In the modified headers, short annotations represent aspects of otherwise complex code. It is also possible to inject code, or include custom header files. An example of these annotations is given in Listing 1. Each annotation set before a function signature instantiates a template with the given name and defines it arguments.

The `siox-wrapper-generator`, a dedicated python tool, either creates code suitable for `ld`'s `--wrap` flag or for use with `LD_PRELOAD`. The tool uses so-called templates to turn annotations into source code, thus a single annotated header allows for many different outputs by switching templates.

```
//@activity
//@activity_link_size fh
//@activity_attribute filePosition offset
//@splice_before ''int intSize; MPI_Type_size(datatype, &intSize);
                  uint64_t size=(uint64_t)intSize*(uint64_t)count;''
//@activity_attribute bytesToWrite size
//@error ''ret!=MPI_SUCCESS'' ret
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void * buf, int count,
                      MPI_Datatype datatype, MPI_Status * status);
```

Fig. 1. Annotations for `MPI_File_write_at()` in the header.

We constantly update the capabilities of the instrumentation. At the moment, the instrumentation covers a number of functions in different I/O interfaces: 74 in POSIX, 54 in MPI, 5 in NetCDF and 18 in HDF5. Instrumentation of open and close functions in NetCDF and HDF5 allows SIOX to relate lower-level I/O to these calls. Asynchronous calls are supported by linking the completion of an operation to its start, but require creation of this link in the instrumented interface. A restriction, in this respect, is that we expect that start and end calls are executed by the same thread, but this restriction will be lifted in the future. Concepts to relate calls across process boundaries, e.g. between I/O client and servers, have been developed but are not being used so far.

3.3 Existing Modules

For the basic modules needed by the low-level API, database and file system back-ends are available. The *ActivityFileWriter* and *StatisticsWriter* store the respective information into a private file, persisting observed activities and statistics without the need to set up a database. There are three alternative *ActivityFileWriter* modules with alternative representations available: A text file using the Boost library, our own binary format and a Berkeley DB implementation. These modules can be embedded in the daemon to all activities of multiple processes in a file or in the process to store its activities independent of other processes. With the *DatabaseTopology*, we have implemented a PostgreSQL database

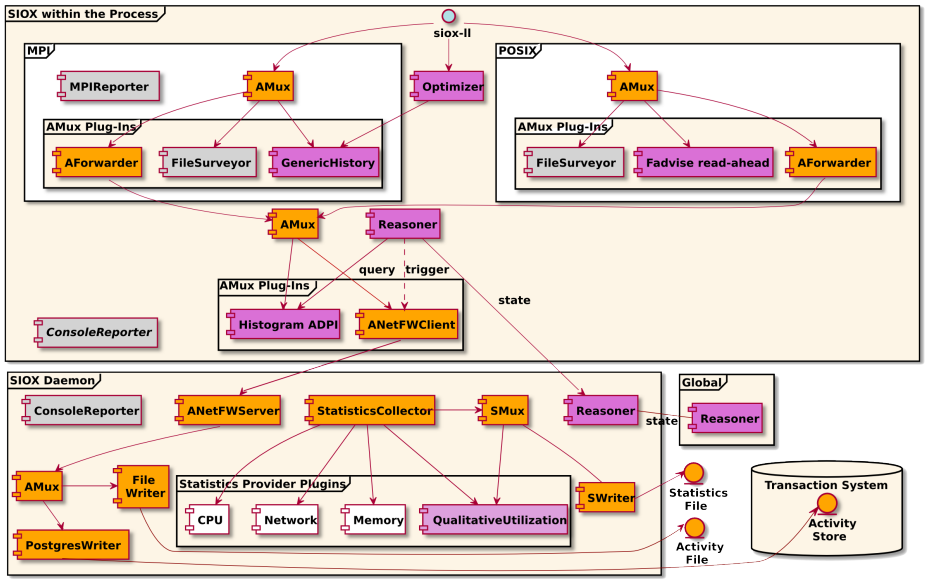


Fig. 2. Example configuration of SIOX modules within a process and the node-local daemon, and their interactions. Those used for monitoring are styled orange or white, those for self-optimization purple; utility modules are gray.

driver for key-value like tables. This module offers a convenient interface for a *TopologyOntology*, *TopologySystemInformation*, and *TopologyAssociationMapper*. Also, the *PostgresWriter* stores the activities into a PostgreSQL database.

The *GIOCommunication* module handles all communication within SIOX. It uses GLIB IO sockets, thus offering both TCP/IP and Unix sockets connections. For each communication partner, two threads are started: one handling incoming messages, and one for transmission.

The typical configuration and interactions of higher-level modules are illustrated in Figure 2. In this setup, a client with POSIX and MPI instrumentation transfers observations to a node-local daemon, which, in turn, injects the activities into the transaction system. Additionally, system statistics are gathered by the daemon. The responsibility of these modules is briefly described in the following:

- *Optimizer*: The current optimizer implementation is very lightweight, dispatching the requests for optimal parameters to plug-ins.
- *ActivityMultiplexerAsync*: This implementation of an activity multiplexer provides both a synchronous processing of completed activities and spawns a thread for background processing of asynchronous notification of registered activity plug-ins. Due to potentially concurrent execution of activities, each plug-in is responsible for protecting critical regions.

- *Reasoner*: Any reasoner will play one of three roles, indicating its scope of responsibility: Process, node, or system (global) reasoner. They periodically poll each AnomalyDetectionPlugIn within their respective domain for an aggregated view of all anomalies witnessed during the last polling cycle. These will be related to the latest generalized health reports of neighboring reasoners in the SIOX hierarchy to form a comprehensive view of the local subsystem's health, In this context, the collected anomalies are evaluated, and the decision is made whether to signal an anomaly to all registered listeners. Although the current standard implementation has a very simple decision matrix with only a few heuristic rules, later versions will play a crucial part in regulating the stream of log data.
- The *ANetFWClient* provides a ring buffer to store a number of recently observed activities. A connected reasoner may emit an anomaly signal which causes the ANetFWClient to transmit all pending activities to a remote *ANetFWServer*. Several configurations are possible; at the moment, the process only sends its data to a daemon if the process-internal reasoner decides to do so. In another configuration, a process may always send its data to the node-local daemon which forwards it if an anomaly is detected at node level.
- *GenericHistory*: A plug-in monitoring accesses and the hint set which was active during their execution. After a learning phase, it can identify the hint set most advantageous to a given operation's performance in the past; these can be further conditioned on user ID and file name extension. The optimizer will query this plug-in, and inject commands to set the hints appropriately before each access; this requires instrumenting the layer issuing the access calls for SIOX. The calls to be observed, and the attributes governing their performance (such as data volume or offset) are configurable.
- *Histogram ADPI*: This plug-in either learns a typical runtime histogram for each type of activity, or it reads the required data from the database. This data is then used to categorize the speed for subsequent activities into very slow, slow, normal, fast, and very fast operations. An aggregate view of this information is supplied to the reasoner, which may then judge the overall system state in turn.
- *FileSurveyor*: Activities of the classes *Read*, *Write*, and *Seek* are monitored here, counting sequential (further distinguished by stride size) and random accesses, and reporting the totals upon application termination. The calls belonging to each class may be configured according to the layer surveyed.
- *FadviseReadAhead*: This plug-in tracks POSIX I/O, and may decide to inject `posix_fadvise()` calls to read-ahead future data. The decision is made based on the amount of data accessed in the previous call, and the spatial access pattern. For each file, it predicts the next access position of the stream, and initiates a call to `fadvise()` if a configurable number of preceding predictions have been correct.
- Statistics infrastructure: Statistics are provided by *StatisticsProviderPlugins*. Their task is to acquire the data, tag it with ontology attributes and topology paths, and to make it accessible to SIOX via a simple interface. A *StastisticsCollector* then polls all the providers registered with it; the current

implementation uses a dedicated thread for this purpose. Afterwards, the collector calls a *StatisticsMultiplexer* to distribute the statistics information to its listeners.
All *Statistic*s can remember values from the near past, going back as far as 100 minutes. As our polling interval for statistics is 100 milliseconds, we cannot store the entire history at full resolution. Consequently, the statistics data is aggregated into longer intervals, providing five different sampling frequencies, each storing its last ten samples, and each serving as the basis to aggregate at the next level. With this approach, it is possible to ensure a reasonable memory overhead while providing long history information. The sampling intervals used in SIOX are 100 milliseconds, 1 second, 10 seconds, 1 minute, and 10 minutes.

– *StatisticsProviderPlugins*: Currently, five StatisticsProviderPlugins are available, collecting information on CPU, memory, network, and I/O load. The fifth plug-in is the QualitativeUtilization plug-in, which acts both as a StatisticsMultiplexerListener and a StatisticsProviderPlugin, integrating the detailed information supplied by the specialized plug-ins into four simple high-level percentages describing the relative utilization of CPU, network, I/O-subsystem and memory.

4 Analysis and Visualization of I/O

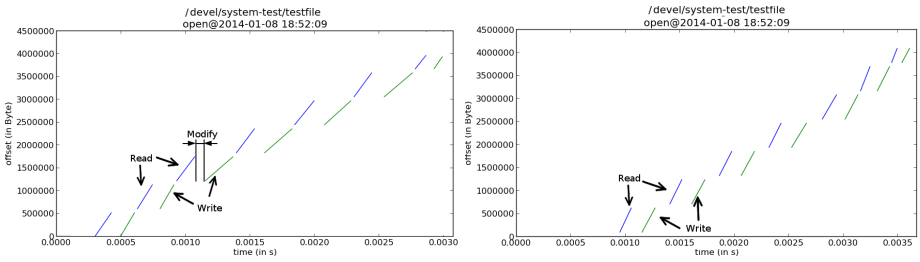
Post-mortem and near-line analysis of observed activity in files and the database are crucial. At the moment, we offer a command-line trace reader and a database GUI. Additionally, each SIOX-instrumented application and the daemon gather statistics about their own usage and overhead. This reporting data is usually output during termination of a process.

4.1 Command-line Trace Reader

The command-line trace reader offers a plug-in interface to process monitored activities. The *print* plug-in just outputs all recorded trace information, replacing attributes with their human-readable representations. An excerpt of a trace is given in Figure 4; note the cause and all potential relations of an activity printed at the end of each line. Another plug-in, the *AccessInfoPlotter* extracts the observed spatial and temporal access pattern for each process and uses pyplot to illustrate the I/O behavior. Figure 3 shows the interleaved I/O of two processes each writing 20×100 KiB blocks with one non-contiguous `MPI_File_write()`. Thanks to data sieving, these patterns lead to several read-modify-write cycles (500 KiB of data per iteration). Both traces have been channeled through a local daemon, and are stored in a single trace file.

4.2 Database GUI

The visualization of the large amount of data produced by a fully functional deployment of SIOX is a challenging and resource-intensive task that exceeds



(a) Rank 0 **(b)** Rank 1

Fig. 3. Generated plots for the observed POSIX access pattern from a shared file accessed by two processes using non-contiguous I/O. Each process locks a file region, reads the data, modifies it and writes it back. Selected phases are marked with arrows to illustrate the behavior.

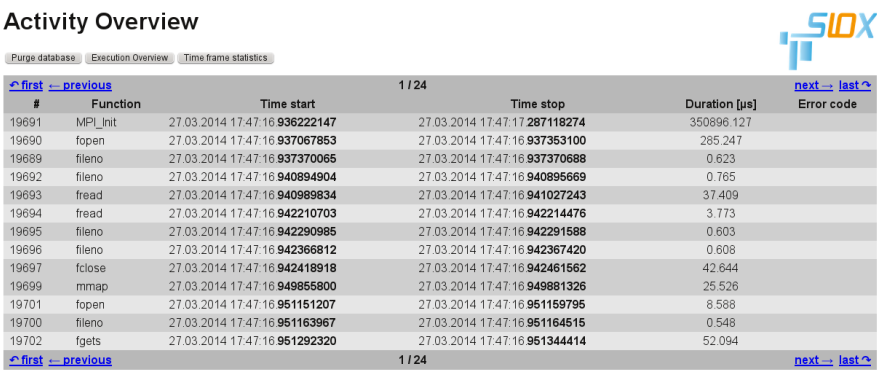
```
0.0006299 ID1 POSIX open(POSIX/descriptor/filename="f1",POSIX/descriptor/filehandle=4) = 0
0.0036336 ID2 POSIX write(POSIX/quantity/BytesToWrite=10240, POSIX/quantity/BytesWritten=
  10240, POSIX/descriptor/filehandle=4, POSIX/file/position=10229760) = 0 ID1
0.0283800 ID3 POSIX close(POSIX/descriptor/filehandle=4) = 0 ID1
```

Fig. 4. Example trace output created by the trace-reader. ID* is the locally generated ID (shortened in this example). The relation between `open()` and the other calls is recorded explicitly.

the scope of the project. However, the possibility of a user-friendly inspection of the data stored in the database is essential for the development and administration of the SIOX system. For this reason, we created a web interface based on HTML/PHP that extracts and presents the information we are interested in. Giving the simplicity of its implementation, this interface is extensible without much programming effort, resulting in a useful tool for experimenting with the collected data as well as for debugging the system. Currently, the interface offers a listing of all activities stored in the database (see Figure 5), as well as a detailed view of any particular activity together with the causal chain of sub-activities it produced (Figure 6).

4.3 Reporting

By way of the Reporter module, any SIOX component may compile a report upon component shutdown. The *ConsoleReporter* module will collect all reports, and write them to the console for later inspection by the user. Report data is structured into groups, and every field can be accessed separately, allowing for further processing; the *MPIReporter* module, for instance, aggregates reports over several nodes, computing a minimum, maximum, and average for every numeric value reported. Figure 7 demonstrates some statistics collected by the FileSurveyor plug-in for a single file during a Parabench [10] run.



Activity Overview

Purge database | Execution Overview | Time frame statistics

↶ first ← previous		1 / 24			next → last ↷
#	Function	Time start	Time stop	Duration [µs]	Error code
19691	MPI_Init	27.03.2014 17:47:16.**936222147**	27.03.2014 17:47:17.**287118274**	350896.127	
19690	fopen	27.03.2014 17:47:16.**937067853**	27.03.2014 17:47:16.**937353100**	285.247	
19689	fileno	27.03.2014 17:47:16.**937370065**	27.03.2014 17:47:16.**937370688**	0.623	
19692	fileno	27.03.2014 17:47:16.**940894904**	27.03.2014 17:47:16.**940895669**	0.765	
19693	fread	27.03.2014 17:47:16.**940989834**	27.03.2014 17:47:16.**941027243**	37.409	
19694	fread	27.03.2014 17:47:16.**942210703**	27.03.2014 17:47:16.**942214476**	3.773	
19695	fileno	27.03.2014 17:47:16.**942290985**	27.03.2014 17:47:16.**942291588**	0.603	
19696	fileno	27.03.2014 17:47:16.**942366812**	27.03.2014 17:47:16.**942367420**	0.608	
19697	fclose	27.03.2014 17:47:16.**942418918**	27.03.2014 17:47:16.**942461562**	42.644	
19699	mmap	27.03.2014 17:47:16.**949855800**	27.03.2014 17:47:16.**949881326**	25.526	
19701	fopen	27.03.2014 17:47:16.**951151207**	27.03.2014 17:47:16.**951159795**	8.588	
19700	fileno	27.03.2014 17:47:16.**951163967**	27.03.2014 17:47:16.**951164515**	0.548	
19702	fgets	27.03.2014 17:47:16.**951292320**	27.03.2014 17:47:16.**951344414**	52.094	
↶ first ← previous		1 / 24			next → last ↷

Fig. 5. Activity list showing I/O function and timestamps.

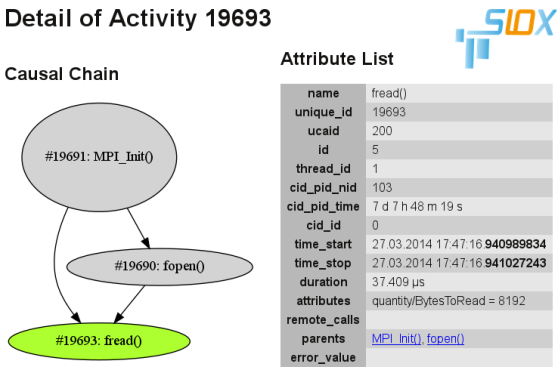


Fig. 6. Detailed view of activity showing the causal chain and list of attributes.

5 Experiments

In this section, we analyze the overhead of the SIOX infrastructure, and discuss two use-cases in which SIOX already improves performance.

5.1 System Configuration

The experimental configuration on the WR cluster consists of 10 compute nodes (2×Intel Xeon X5650@2.67GHz, 12 GByte RAM, Seagate Barracuda 7200.12) and 10 I/O nodes (Intel Xeon E3-1275@3.40GHz, 16 GByte RAM, Western Digital Caviar Green WD20EARS) hosting a Lustre file system. A dual-socket compute node provides a total of 12 physical cores and 24 SMT cores. All nodes are interconnected with gigabit ethernet, thus, the maximum network throughput between the compute and the I/O partition is 10 × 117 MB/s. As a software

```
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Accesses = (40964,40964,40964)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Accesses/Reading/Random, long seek = (20481.8,20480,20482)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Accesses/Reading/Random, short seek = (0,0,0)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Accesses/Reading/Sequential = (0.2,0,2)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Bytes = (8.38861e+09,8.38861e+09,8.38861e+09)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Bytes/Read per access = (204780,204780,204780)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Bytes/Total read = (4.1943e+09,4.1943e+09,4.1943e+09)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Seek Distance/Average writing = (1.0238e+06,1.0238e+06,1.02382e+06)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Time/Total for opening = (3.9504e+08,3.66264e+08,4.38975e+08)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Time/Total for reading = (1.47169e+11,1.0968e+11,1.76617e+11)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Time/Total for writing = (1.08783e+12,1.03317e+12,1.16192e+12)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Time/Total for closing = (1.0856e+11,6.11782e+10,1.46834e+11)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Time/Total surveyed = (1.34568e+12,1.34568e+12,1.3457e+12)
```

Fig. 7. Example report created by FileSurveyor and aggregated by MPIReporter (shortened excerpt). The number format is (average, minimum, maximum).

basis, we use Ubuntu 12.04, GCC 4.7.2, and OpenMPI 1.6.5 with ROMIO. For the overhead measurement, we compiled SIOX using `-O2`.

5.2 Instrumentation of Concurrent Threads

The software instrumentation of SIOX adds overhead to the critical path of applications. To assess this overhead, we created a multi-threaded benchmark: each thread calls `fwrite()` without actually writing any data. Without instrumentation, a single call needs roughly 6 ns or 14 CPU cycles. The overhead of the instrumented `fwrite()` is visualized in Figure 8 for 1 to 24 threads and three different configurations. *SIOX plain* refers to an almost empty configuration without additional modules, in *SIOX POSIX fw* we add a configuration section for POSIX containing the AForwarder and the additional AMux for POSIX. In the *SIOX process* configuration, we enabled all the modules inside a process as shown in Figure 2 (the Reasoner is not included here because it contains only a low-frequency periodic thread).

Since we must protect critical regions for each module, an increasing number of threads compete for these resources, leading to contention. In our measurements, the critical section accounts for a runtime of about 0.75 μs (plain configuration) to about 6 μs (process configuration). As every activity is intended to be transferred to the node daemon, if an anomaly occurs, this path may impose a bottleneck. A benchmark of the communication module demonstrates a sustained transfer rate of 90,000 messages/s (1 KiB payload). Thereby, it should handle typical I/O scenarios. The file writer modules store activities with a rate of 70,000 activities/s, thus they could persist the activities on node level.

We did not include the database back-end for activities in our measurements because our PostgreSQL instance on a VM just inserts about 3,000 activities/s and thus is a bottleneck we are working on. Regardless of the low integration of activities into our transaction system; as these processes are usually done by the daemon, they happen concurrent to the application execution not deferring application I/O.

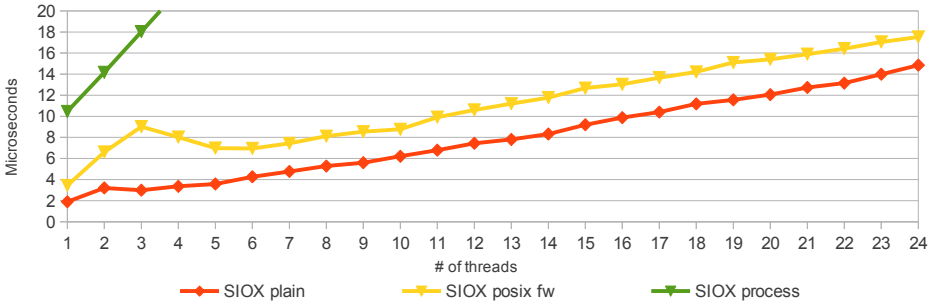


Fig. 8. Overhead per thread due to critical regions in the modules.

5.3 Instrumentation of the ICON Climate Model

In addition to our benchmark experiments, we also measured the impact of SIOX on the runtime of a climate model. For this test, we used the ICON model [11] developed by the Max Planck Institute for Meteorology (MPI-M) and the German Weather Service (DWD). The test setup was as follows: First we simulated one day using a 20480 cell icosahedral grid (ICON's R2B04 grid), utilizing all 12 physical cores available on one cluster node. This takes 100.7 seconds on average. Then the model was rerun with different levels of SIOX instrumentation using `LD_PRELOAD`. This was repeated ten times to get runtime measurements that were precise enough to be interpreted.

The resulting times show an overhead between 2.5 and 3.0 seconds when only POSIX or MPI were instrumented and 5.0 seconds for both. Measurement errors ranged from 0.29 seconds to 0.65 seconds. To assess the amount of relative and absolute overhead, the entire test was repeated with twice the simulation time which takes 193.3 seconds on average without instrumentation. In this test, only the overhead of the pure MPI instrumentation increased to 4.5 seconds, the other two instrumentation overheads increased slightly but remained within their respective error intervals.

Much of the constant overhead is due to the reporting output produced by SIOX or by the initialization of the database connection. The incremental costs of SIOX, however, are barely measurable.

5.4 Instrumentation/Optimization of Parabench: 4 Levels of Access

In this experiment, we instrument the parallel I/O benchmark Parabench with SIOX. Additionally, we sketch a scenario in which SIOX will improve performance by automatically setting MPI hints.

In our strided access pattern, each process accesses 10240 blocks with a size of 100 KiB, which accumulate in a shared file of 10 GiByte. We measure the performance of the four levels of access in MPI. According to [12], they are defined by the two orthogonal aspects: collective vs. independent I/O, and contiguous

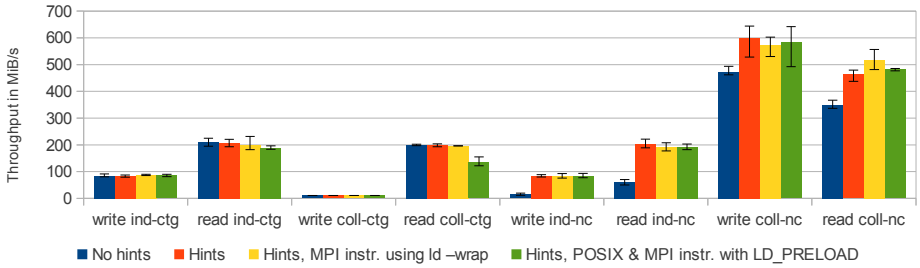


Fig. 9. Performance comparison of the 4-levels of access on our Lustre file system. The configuration with hints increases the collective buffer size to 200 MB and disables data sieving.

vs. non-contiguous I/O. The observed performance is illustrated in Figure 9, the four configurations are as follows: First, there is a configuration without user-supplied hints, the second configuration adds hints but no SIOX instrumentation, the third adds MPI instrumentation using SIOX's `ld` wrap option, and the last uses `LD_PRELOAD` to instrument both, MPI and POSIX.

On our system, we observe that data-sieving decreased performance significantly (look at the ind-nc cases), therefore we disable it in our hint set.

Overall, the observable performance of the instrumented Parabench is comparable to a regular execution. There is one exception; at 130 MiB/s, the read coll-ctg is behind the normally measured performance of 200 MiB/s. Thanks to our reporting, we could understand the cause. Since we intercept all POSIX I/O operations, the socket I/O caused by MPI is also monitored, adding overhead to each communication[4].The MPI communication causes 650,000 activities per process and is most intense for collective contiguous I/O. In total, I/O accounts to just about 60,000 and 41,000 activities, for POSIX and MPI, respectively. Also, the FadviseReadAhead applies read-ahead hints to socket I/O (without this plug-in, performance improves to 170 MiB/s).

A potential gain for users will be the automatic learning of SIOX, which we just started to explore with the first plug-ins. By virtue of the GenericHistory plug-in, SIOX can already automatically set hints during `MPI_File_open` that have proven to be beneficial in the past. If we modify our benchmark slightly to repeat the test with different hint sets (for example, the default hints, and the improved ones), then the plug-in will remember the improved hints, and set all hints for subsequent opens that do not define them. Thus, without any modification or recompilation of the application, users would benefit from globally known hints.

[4] With the newest version, is possible to report these operations to a "POSIX_Network" component, handling them differently to I/O operations. However, this is not done in this benchmark.

Experiment	20 KiB stride	1000 KiB stride
Regular execution	97.1 μs	7855.7 μs
Embedded fadvise	38.7 μs	45.1 μs
SIOX fadvise read-ahead	52.1 μs	95.4 μs

Table 1. Time needed to read one 1 KiB data block in a strided access pattern.

5.5 Read-ahead with fadvise

With the FadviseReadAhead plug-in, a module has been implemented which detects a strided read access pattern and injects `posix_fadvise()` to fetch data for the next access – if the last 4 predictions have been correct. To assess its performance, a small benchmark is created which loops over 10 GiB of data stored on a compute node's local disk. On each iteration, the benchmark simulates compute time by sleeping a while, then it seeks several KiB forward and reads a 1 KiB chunk – the whole area covered by the seek and one access is defined as the stride size. Two different strides are evaluated: 20 KiB and 1000 KiB. Sleep time is adjusted from 100 μs in the 20 KiB case to 10 ms to make read-ahead possible. The benchmark is executed several times, between each run the page cache is cleared using `echo 3 > /proc/sys/vm/drop_caches`; the deviation between runs is below 1% of runtime.

In order to validate the results, the `posix_fadvise()` calls issued by the SIOX module have also been embedded into the original source code, thus yielding best performance without any overhead from SIOX. Table 1 compares regular, uncached execution with the manual source code modifications and the FadviseReadAhead module. It can be observed that fadvise improves performance already for 20 KiB strides but excels at 1000 KiB stride, decreasing time per I/O from 7.8 ms to 45 μs. This improvement can be explained by the data placement: Since EXT4 tries to place logical file offsets close together on the drive's logical block addressing, the larger stride forces movement of the disk's actuator. Consequently, with appropriate hints, the programmer can reduce I/O time drastically, but this requires manual adaption of the code. The module shipped with SIOX adds some overhead but improves performance similarly and automatically.

6 Future Work

Opportunities for future work abound. With our basic infrastructure, we can now quickly develop new modules dedicated to certain purposes. Besides new optimizations, we are working on improving the performance of our transaction system which, in turn, we will use to record a large training set of I/O from various benchmarks and applications. Once available, we can improve the rules of our reasoner module, and evaluate machine learning techniques to extract

further knowledge. Also, new services will be located in the daemon to cache the necessary information of the global services, such as the ontology.

In the consortium, we are currently working on a first prototype for a GPFS plug-in in OpenMPI which is explicitly instrumented for SIOX. The implementation will also monitor GPFS using the Data Management API framework[13] that is used to keep track of I/O events for selected files and file regions. The SIOX high-level I/O library is designed to not only monitor I/O operations unidirectionally, it is also capable of receiving optimization hints from SIOX at runtime. Finally, we aim to automatically select the best GPFS hints based on the utilization and historical I/O records from our knowledge bases. However, the various modules imaginable for pattern creation and matching, for anomaly detection and for optimization, as well as the machine learning algorithms, offer a rich field for researchers and system administrators alike.

7 Summary and Conclusions

Our vision for SIOX is a system that will collect and analyze activity patterns and performance metrics in order to assess and optimize system performance. In this paper, we presented results of our prototype and have given a glimpse of its flexible and modular architecture. Although the monitoring with SIOX imposes some overhead in the critical path of I/O, we were able to demonstrate that this overhead is very small compared to the performance gains that may be achieved. The I/O stack offers a large variety of potential optimizations which need to be controlled intelligently, and the SIOX system provides the architecture to do so. In this sense, SIOX is the swiss army knife for experimenting with alternative and automatic I/O optimizations without even modifying existing code. Since the overhead is bearable, we believe that (with appropriate modules and configuration) constant supervision with SIOX will greatly improve performance in data centers.

References

1. Carns, P.H., Latham, R., Ross, R.B., Iskra, K., Lang, S., Riley, K.: 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 (September 2009)
2. Madhyastha, T., Reed, D.: Learning to Classify Parallel Input/Output Access Patterns. Parallel and Distributed Systems, IEEE Transactions on **13**(8) (August 2002) 802–813
3. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using Magpie for Request Extraction and Workload Modelling. Proceedings of the 6th Symposium on Opearting Systems Design and Implementation **6** (2004) 259–272
4. Yuan, C., Lao, N., Wen, J.R., Li, J., Zhang, Z., Wang, Y.M., Ma, W.Y.: Automated Known Problem Diagnosis with Event Traces. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006. EuroSys '06, New York, NY, USA, ACM (2006) 375–388

5. Sandeep, S.R., Swapna, M., Niranjan, T., Susarla, S., Nandi, S.: CLUEBOX: a Performance Log Analyzer for Automated Troubleshooting. In: Proceedings of the First USENIX conference on Analysis of system logs. WASL'08, Berkeley, CA, USA, USENIX Association (2008)
6. Duan, S., Babu, S., Munagala, K.: Fa: A System for Automating Failure Diagnosis. In: Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on. (29 2009-April 2 2009) 1012–1023
7. Behzad, B., Huchette, J., Luu, H.V.T., Aydt, R., Byna, S., Yao, Y., Koziol, Q., Prabhat: A framework for auto-tuning hdf5 applications. In: Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing. HPDC '13, New York, NY, USA, ACM (2013) 127–128
8. Wiedemann, M.C., Kunkel, J.M., Zimmer, M., Ludwig, T., Resch, M., Bönisch, T., Wang, X., Chut, A., Aguilera, A., Nagel, W.E., Kluge, M., Mickler, H.: Towards I/O Analysis of HPC Systems and a Generic Architecture to Collect Access Patterns. Computer Science - Research and Development **1** (2012) 1–11
9. Zimmer, M., Kunkel, J., Ludwig, T.: Towards Self-optimization in HPC I/O. In Kunkel, J.M., Ludwig, T., Meuer, H.W., eds.: Supercomputing. Number 7905 in Lecture Notes in Computer Science, Berlin, Heidelberg, Springer (06 2013) 422–434
10. Mordvinova, O., Runz, D., Kunkel, J., Ludwig, T.: I/O Performance Evaluation with Parabench – Programmable I/O Benchmark. Procedia Computer Science (2010) 2119–2128
11. Max-Planck-Institut für Meteorologie: ICON. `http://www.mpimet.mpg.de/en/science/models/icon.html`
12. Thakur, R., Gropp, W., Lusk, E.: Optimizing Noncontiguous Accesses in MPI/IO. Parallel Computing **28**(1) (2002) 83 – 105
13. IBM: Data Management API Guide. (2013)