A Methodology for Performance Analysis of Applications Using Multi-layer I/O

Ronny Tschüter(✉), Christian Herold, Bert Wesarg, and Matthias Weber

Center for Information Services and High Performance Computing, Technische Universität Dresden, Dresden, Germany
ronny.tschueter@tu-dresden.de

Abstract. Efficient usage of file systems poses a major challenge for highly scalable parallel applications. The performance of even the most sophisticated I/O subsystems lags behind the compute capabilities of current processors. To improve the utilization of I/O subsystems, several libraries, such as HDF5, facilitate the implementation of parallel I/O operations. These libraries abstract from low-level I/O interfaces (for instance, Posix I/O) and may internally interact with additional I/O libraries. While improving usability, I/O libraries also add complexity and impede the analysis and optimization of application I/O performance. In this work, we present a methodology to investigate application I/O behavior in detail. In contrast to current methods, our approach explicitly captures interactions between multiple I/O libraries. This allows to identify inefficiencies at individual layers of the I/O stack as well as to detect possible conflicts in the interplay between layers. We implement our methodology in an established performance monitoring infrastructure and demonstrate its effectiveness with an I/O analysis study of a cloud model simulation code. In summary, this work provides the foundation for application I/O tuning by exposing inefficiency patterns in the usage of I/O routines.

Keywords: I/O · Performance analysis · Monitoring
Instrumentation

1 Introduction

Modern HPC systems provide powerful storage hardware equipped with high bandwidth interconnects and parallel file systems. Nevertheless, input and output (I/O) operations still present a major limitation factor for the performance of scientific applications. Current research topics, such as *big data* and *machine learning*, further increase the trend of processing large data volumes.

Highly-scalable applications transfer data in parallel to cope with large data volumes and efficiently utilize available I/O resources. A wide range of I/O libraries, such as HDF5 [24], NetCDF [26], and MPI I/O [18, Chap. 13] support developers in implementing parallel I/O operations by abstracting from

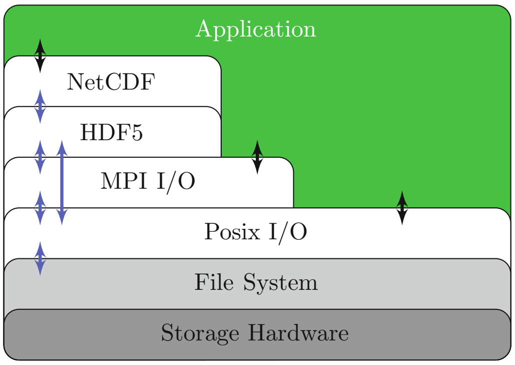


Fig. 1. Software layers of an application using three I/O interfaces concurrently.

low-level I/O interfaces. Often, these libraries provide features for storing meta-data to describe the data format and units along with specific data values. This further increases the data volume in addition to the actual raw data.

Hiding the complexity of implementing low-level parallel I/O operations is a major benefit of I/O libraries. Yet, using I/O libraries does not necessarily guarantee efficient I/O resource utilization [10]. Improved usability gained by abstraction also implies a more challenging I/O performance analysis. This is especially true for applications using multiple I/O interfaces concurrently. Figure 1 shows an example application that uses multiple I/O libraries independently. The application itself calls NetCDF, MPI I/O, and Posix I/O functions directly. The NetCDF library issues HDF5 function calls. HDF5 in turn contains MPI I/O and Posix I/O in its software stack. Complex interactions between I/O libraries and user code impact each other. It is essential to gather information from all involved I/O layers to evaluate the effectiveness of resulting I/O operations. This allows detailed understanding of the actual I/O behavior and enables the identification of underlying root causes of I/O problems. For example, I/O operations are propagated through the I/O software stack. An `open` call at the top level will also cause `open` operations in lower levels. Hence, each layer of the I/O software stack maintains own file descriptors to manage I/O resources. In case of writing data, each I/O layer may rearrange operations or add additional meta-information to the actual raw data. Thus, to correctly assign and evaluate specific operations, we need to capture information at each individual I/O layer.

Monitoring of multi-level I/O operations poses two challenges: (a) recording I/O operations arising from multiple I/O libraries and (b) recording of interactions between individual I/O libraries. This work addresses both challenges. Thereby, we support users in investigating and improving the I/O performance of parallel applications. Our contributions are:

– An approach to record information about I/O resources used by applications as well as performance relevant data of I/O operations including the interactions of multiple I/O libraries.

– Tracking the mount information of I/O resources in order to determine their generic scope and recording this information for enhanced analyses.
– Implementation of the approach in an established monitoring infrastructure.
– A detailed I/O analysis study of a real-world application to demonstrate the applicability of our approach.

2 Related Work

Several techniques exist for monitoring I/O activities. In principle these approaches can be distinguished by: (a) the data acquisition scope (system or application) (b) the recorded data format (statistics or event log) and (c) the ability for monitoring relations between individual I/O layers.

Statistics on System-Level: The tools iotop [15], iostat [14], blktrace [5], and sar [20] monitor system performance with special focus on I/O resource usage. These tools collect statistics and report measurement values per device, partition, or network filesystem as well as a global view of the whole system.

Statistics on Application-Level: Arm MAP [4], Darshan [8], and TAU [22] monitor individual applications. Among other runtime events, like function entry and exit, they can record information about I/O operations. With respect to I/O, Arm MAP focuses on Posix I/O and captures Lustre [16] counters, whereas Darshan and TAU record Posix I/O and MPI I/O activities. HPCToolkit [1] intercepts selected I/O operations and records their number of bytes read or written to mark I/O intensive application phases. In contrast to our work, all of the previously mentioned tools collect statistics.

Event Logs on Application-Level: VampirTrace [19] records I/O activities and writes the collected information to event logs. However, it only records calls to I/O functions of the standard C library and is no longer supported. Its successor Score-P [11] does not support I/O recording yet. ScalaIOTrace [28] generates compressed event logs of MPI I/O and Posix I/O function calls. None of the mentioned tools explicitly correlates individual layers of the I/O software stack.

Visualization: Vampir [2] visualizes event logs generated by VampirTrace and Score-P in timeline and statistical charts. Event logs retain temporal information of each individual event. This enables detection of performance problems with changing characteristics over application runtime. The Virtual Institute for I/O (VI4IO) [27] is a collaboration platform for research groups in the field of HPC I/O. It provides an overview about I/O middleware, benchmarks, and tools.

3 Methodology

This section describes our approach for analyzing applications using multiple I/O libraries. We cover both I/O resources (e.g., files and file descriptors) as well as I/O activities (e.g., reading and writing). Therefore, we distinguish between **definitions** and **events**. *Definitions* provide detailed information about I/O resources, whereas *events* represent I/O activities during application runtime.

3.1 Definitions

Definitions describe resources of I/O operations. Posix I/O operations do not directly work on input/output resources, but use file descriptors as an abstract handle. This allows multiple processes/threads to access the same file independently. Consequently, our definitions, Fig. 2, distinguish between I/O resources and file descriptors. The following paragraphs introduce each definition in detail.

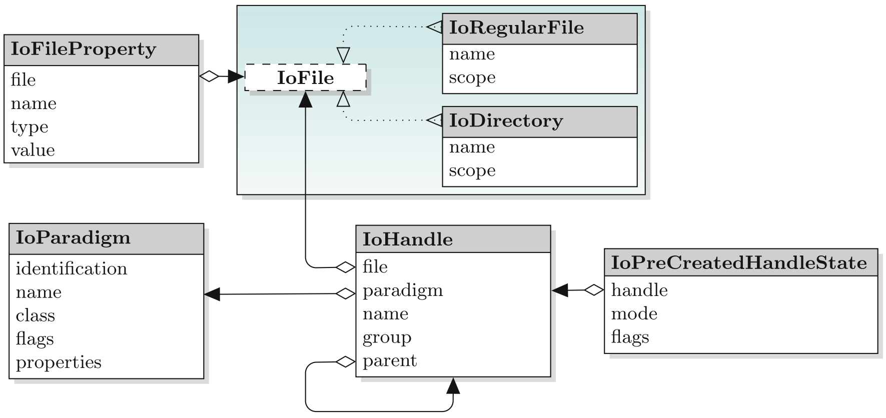
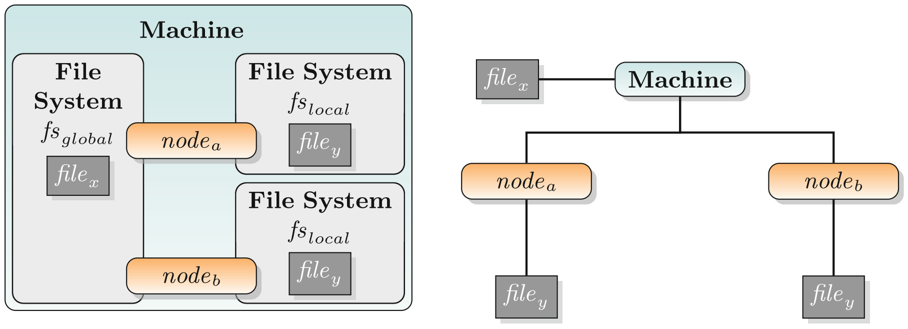


Fig. 2. Overview of definitions to reflect I/O resources and their relationships.

Definitions of I/O Resources: According to the "*Everything is a file*" philosophy Unix and its derivatives treat a wide range of I/O resources as a file, e.g., files, directories, and sockets. This is reflected by the polymorph **IoFile** definition that provides a common namespace for objects used by I/O operations. Currently, definitions for files (**IoRegularFile**) and directories (**IoDirectory**) are available within this namespace. However, it is possible to add further definitions to this namespace.

IoRegularFile and *IoDirectory* definitions store the `name` of a file or directory. HPC machines mount several file systems concurrently. Thus, name or path alone do not represent unique identifiers for I/O resources. In principle, two categories of file systems can be distinguished: (a) local file systems available only on a single compute node (b) global file systems shared via network on the whole machine. Figure 3 depicts an example. The illustrated compute nodes $node_a$ and $node_b$ use two different file systems—a shared network file system fs_{global} and a local scratch file system fs_{local}. The file $file_x$ in fs_{global} is accessible on the whole machine. In contrast $file_y$ represents two distinct physical files, because they reside in separate file systems fs_{local}. Therefore, the `scope` attribute marks the physical scope with regard to the system topology.

The **IoFileProperty** definition attaches user-defined attributes (e.g., mount point information or Lustre strip policy) to an *IoFile* definition.



(a) Hardware topology (b) System tree representation

Fig. 3. A file's scope depends on its storage position (global or local file system).

Definitions of I/O Handles: IoParadigm describes an available I/O library. The `identification` attribute categorizes an *IoParadigm* (e.g., "MPI I/O"), while the `name` distinguishes specific implementations (e.g., "OMPIO" [9] or "ROMIO" [23]). The `class` attribute specifies whether the I/O paradigm is serial or parallel. Only parallel I/O paradigms enable collective I/O operations within a group of multiple processes/threads. The `flags` attribute allows to set further boolean characteristics for the I/O paradigm (e.g., mark if the paradigm either directly accesses the operating system or maps its functionality to other I/O paradigms such as HDF5 or NetCDF). In addition, *IoParadigm* provides an extensible mechanism to specify further `properties` such as version information.

An **IoHandle** definition reflects a file descriptor based on a prior I/O resource definition specified by the `file` attribute. The `parent` attribute of an *IoHandle* models hierarchical relations between I/O handles. This mechanism enables correlation of operations between individual layers of the I/O software stack. If the `paradigm` supports collective I/O operations, the `group` attribute specifies the set of participating processes/threads.

The **IoPreCreatedHandleState** definition marks a `handle` that is standardly created (e.g., `stdin`, `stdout`, `stderr`) or inherited from a parent process/thread. The definition holds the access `mode` (e.g., read or write) and status `flags` of this default I/O handle.

3.2 Events

Events represent I/O activities at application runtime. In this work, we focus on events required for performance analysis. Therefore, we assume that I/O operations finish successfully, otherwise performance analysis is not reasonable. However, our approach is not limited to performance analysis and we plan to support the handling of unsuccessful I/O operations (see Sect. 7). We distinguish events into meta data (e.g., `open/create`, `close`) and data transfer (e.g., `read/write`) operations. All events store an accurate timestamp and information about the issuing process/thread. Additional information depend on the specific event type.

Meta Data Operations: Events of this category indicate the creation and the destruction of file descriptors. The **IoCreateHandle** event marks the creation of a new `handle` (e.g., after opening a file). The `mode` attribute determines the access mode to the file descriptor (e.g., read-only, write-only, or read-write). According to the Posix I/O API, *IoCreateHandle* stores optional `creationFlags` (e.g., create if the file does not exist) and `statusFlags` (e.g., open file in append mode). An **IoDestroyHandle** marks the end of an active I/O `handle`'s lifetime. Thus, a pair of consecutive *IoCreateHandle* and *IoDestroyHandle* events defines the time in which the handle is active and can be used by other events. The **IoDuplicateHandle** event represents the duplication of an existing file descriptor. This event references the original file descriptor (`oldHandle`) as well as the newly created one (`newHandle`). The *IoDuplicateHandle* activates the `newHandle` and the `oldHandle` remains active. In our event design, the new handle does not inherit the status flags. Instead, the `statusFlags` attribute explicitly records this information. This option releases analysis tools from the need of tracking the inheritance. Figure 4 illustrates the life cycle of tracked I/O handles.

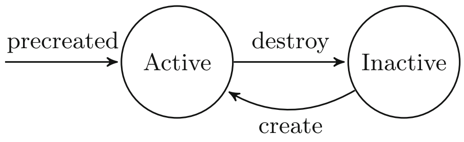
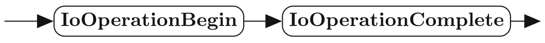


Fig. 4. Events create and destroy I/O handles at runtime. Commands to duplicate handles build a special case: the original handle remains in active state, the newly created handle changes from inactive to active state.

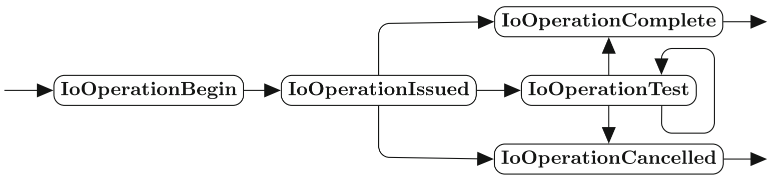
The following events track the status of active I/O handles. The **IoSeek** event records changes of the position within a file. The event stores the offset requested by the user (`offsetRequest`), the position to which the offset should be applied (`whence`), e.g., absolute from the start or end, relative to the current position, and the resulting offset relative to the beginning of the file (`offsetResult`). An **IoChangeStatusFlags** event tracks changes to the status flags of an active `handle`. The `statusFlags` attribute holds the updated status.

The **IoDeleteFile** event marks the deletion of an I/O resource. Similar to deletion functions, such as `unlink`, `rename`, or `remove`, this event operates on I/O resources instead of I/O handles. In addition to the affected `file`, this record stores the `paradigm` that issued the deletion.

Data Transfer Operations: Events of this category record data transfer operations. One complete transfer operation might be split into basic events. Further, we distinguish between blocking and non-blocking operations. For example, a blocking Posix I/O `read` operation consists of two events—one for its start and



(a) Event sequence of blocking I/O operations.



(b) Event sequence of non-blocking I/O operations.

Fig. 5. Sequence of generated events for different I/O operation types.

one for its completion. Consequently, both events need an identifier to relate all parts composing an I/O operation. Therefore, these events contain a `matchingId` attribute, identifying an I/O operation in-flight. The attribute is valid for a process including all its threads. The **IoOperationBegin** event lists the affected `handle`, the operation `mode` (e.g., reading or writing), and `operationFlags` providing additional semantic information. In particular, the `operationFlags` attribute defines two distinct characteristics of an operation: (a) collective or non-collective, and (b) blocking or non-blocking. The `bytesRequest` attribute reflects the user defined maximum number of transferred bytes. An **IoOperationComplete** event marks the end of a data transfer operation. It references the affected `handle`. The `bytesResult` attribute stores the actual number of transferred bytes. Corresponding *IoOperationComplete*-*IoOperationBegin* event time stamps define the transfer operation duration. Figure 5a shows the event sequence generated by blocking I/O data transfer operations. The "blocking" bit in the `operationFlags` of the *IoOperationBegin* event is set accordingly. The semantic of blocking operations ensures that a pair of matching *IoOperationBegin* and *IoOperationComplete* events occurs within the event stream of the same thread. In contrast, Fig. 5b illustrates the event sequence of a non-blocking I/O data transfer operation (e.g., `aio_write`). Typical for non-blocking operations is the decoupling of issuing and completing operation, i.e., started on one thread but completed on another thread of the same process. Non-blocking data transfer operations also start with *IoOperationBegin* events. In case of a successful initiation an **IoOperationIssued** event follows. *IoOperationBegin* and its corresponding *IoOperationIssued* event must occur on the same thread. Users can test active non-blocking operations to ensure their completion. **IoOperationTest** events represent unsuccessful tests (I/O operation not finished yet), *IoOperationComplete* events indicate finished operations. The **IoOperationCancelled** event represents the successful cancellation of a non-blocking operation. Any thread of the same process can test, cancel, or complete a non-blocking I/O operation in-flight.

Collective I/O operations are executed by all processes/threads of the respective I/O handle. The "collective" bit in the `operationFlags` attribute of the *IoOperationBegin* event marks the special semantic of such operations.

4 Implementation

In the previous Sect. 3, we presented our approach for recording I/O operation information, whereas we focus on the implementation details in this section. We implement our design in OTF2 (Open Trace Format Version 2) [12]. Many analysis tools, such as Vampir and Scalasca [13], process OTF2 event traces. The OTF2 library provides an API for reading and writing event traces. It already supports events for function entry and exit, parallelization constructs, and communication. In this work, we extend OTF2 with definitions and events implementing the I/O operations presented in Sect. 3. OTF2 maintains a list of parallelization paradigms (e.g., MPI, OpenMP, Pthreads) as a C-enumeration in its application programming interface[1]. Adding support for new parallelization paradigms would require to extend this enumeration as well. However, this could result in inconsistencies due to unknown enumeration members, when older OTF2 versions read event logs written by a newer OTF2 version. Considering the wide range of available I/O interfaces, we conclude that this approach is unsuitable. Therefore, we abstain from providing a fixed list of supported I/O paradigms in our implementation. Instead, we implement the *IoParadigm* definition record using a self-describing mechanism. For the sake of convenience, the OTF2 library maintains a list of known I/O paradigms in its documentation[2]. Users are encouraged to follow these suggestions when generating their own event logs.

Besides OTF2, we require a software component that monitors the application behavior at runtime. For this purpose, we select the Score-P measurement infrastructure and add components for intercepting calls to specific I/O libraries. In order to intercept calls to MPI, we utilize the existing MPI profiling interface (PMPI) [18, Sect. 14.2]. For all remaining I/O interfaces we use a generic interception method [6]. Each time an application issues an I/O function, we intercept this call. The control flow passes to the Score-P measurement system which has access to all function parameters and can record performance relevant data. Then, the measurement system calls the original function. After the original function returns, the control flow passes back to the application and the program execution continues.

We strive to support a flexible list of I/O paradigms in Score-P. Therefore, Score-P must handle the interactions of I/O paradigms in a generic way. Especially, the mapping of I/O operations to an a priori known lower-level I/O paradigm requires a paradigm agnostic implementation. We achieve this by implementing a shared per-thread I/O management stack. Individual paradigms

[1] https://silc.zih.tu-dresden.de/otf2-2.1.1/OTF2__GeneralDefinitions_8h.html#aa14d0751354081d258913145a80e79a9.

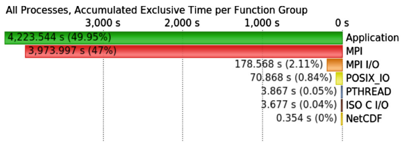
[2] https://silc.zih.tu-dresden.de/otf2-2.1.1/group__io.html.

can communicate via this stack. The following describes this approach using the example case of MPI I/O implemented on-top of ISO-C. If the MPI I/O component from Score-P intercepts a call to `MPI_File_open`, it creates a new *IoHandle* ($handle_1$) and pushes it to the I/O management stack. Then, the `PMPI_File_open` function is called via the MPI profiling interface. The MPI implementation may than call `fopen`, which is subsequently intercepted by Score-P as well. The ISO-C component inspects the top element of the I/O management stack to determine whether a potential higher-level I/O paradigm is active. If a handle is available on the stack ($handle_1$ in this example), this handle is used as parent for the newly created *IoHandle* ($handle_2$). After leaving `fopen` and `MPI_File_open`, the top element from the I/O management stack is removed for each involved paradigm. In summary, whether lower-level paradigms will create new *IoHandle*s is unknown a priori. Therefore, each I/O component must push and pop its current active handle onto the I/O management stack. This ensures proper references to controlling higher-level I/O paradigms in individual handles. As a result, all occurring *IoHandle*s create a root-directed tree.

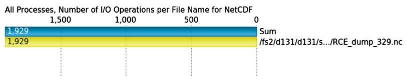
5 Case Study

We evaluate our methodology and implementation in an analysis of the Met Office NERC Cloud model (MONC) simulator. Our study checks MONC for I/O performance penalties and exposes insights of operations using multiple I/O layers. MONC, a Fortran+MPI code, utilizes NetCDF to write results to disk. The cloud simulator has two kinds of processes: (a) simulation processes for computing the cloud model and (b) I/O server processes for storing results to disk. Users can individually set the number of I/O server processes. At runtime, the I/O servers keep simulation results in main memory. After N simulation steps or at program termination, the I/O servers flush the results to disk [7].

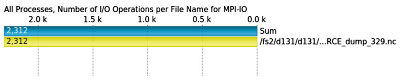
We record the I/O behavior of MONC using our Score-P prototype. Score-P instruments the source code and intercepts library calls to Posix I/O, MPI I/O, and NetCDF. We conduct our experiments on ARCHER [3]. This Cray XC30 system consists of 4920 compute nodes, each containing two 12-core E5-2697 v2 (Ivy Bridge) processors running at 2.7 GHz. Our experiments use a 4.4 PB Lustre file system (stripe count 1, stripe size 1 GiB) to store simulation results and collected event logs. We run MONC on 112 processes, distributed over 8 nodes. Each node hosts one I/O server process with a pool of 10 additional threads. The remaining 104 simulation processes compute the cloud model. In our experiment setup, MONC simulates 100 timesteps. At the end of the application run, the I/O server processes write the data to disk via calls to NetCDF. Using our approach, we can inspect internal function invocations of MPI I/O and Posix I/O. In order to avoid interference with the I/O behavior of the observed application, we keep all collected performance data in main memory during application runtime. After the application has finished, event logs are written to disk. In our experiments the recording of performance data caused an increase in application runtime of about six percent. We visualize the resulting event logs using the tool



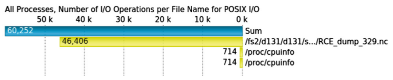
(a) Function summary over all processes shows few time spent in I/O operations.



(b) NetCDF



(c) MPI I/O



(d) Posix I/O

Fig. 6. Function and I/O statistics of the MONC experiment run.

Vampir. Since version 9.4 Vampir features new displays with special focus on the visualization of I/O behavior of applications. The paper "Visualization of Multi-layer I/O Performance in Vampir" [17] presents a detailed description of these sophisticated visualization techniques using I/O related performance data.

Figure 6a depicts the overall exclusive time spent in particular function groups. The event log contains 7 groups, while most of the time is spent in application code (about 50%). Furthermore, the simulator spends more time in MPI communication routines than in I/O operations. Although this first analysis suggests that MONC does not exhibit poor I/O performance it is worth taking a closer look at I/O operations.

To investigate I/O performance in detail, Fig. 6 depicts three I/O summary charts for NetCDF (Fig. 6b), MPI I/O (Fig. 6c), and Posix I/O (Fig. 6d), respectively. All three layers utilize the same `RCE_dump_329.nc` file. The number of accesses to this file increases while traversing the NetCDF, MPI I/O, and Posix I/O layer. This statistic reflects how each library abstracts functionality in order to hide complex operations. Furthermore, the figure shows that Posix I/O also utilizes additional files. In further analyses we will identify the origin of these file accesses.

Figure 7 depicts the I/O timeline (top) and the process summary (bottom) for Thread 7 of Rank 0. The I/O timeline displays the performed type of I/O operations (Read (orange), Write (yellow), Open (blue), Close (green)) on the x-axis and the accessed files as well as associated handles on the y-axis. If an I/O library (e.g., NetCDF) utilizes another I/O library, the individual handles of each library are attached to each other, as represented in a tree-like hierarchy to the left of the upper chart. The top chart in Fig. 7 depicts all handles used to access the NetCDF file `RCE_dump_329.nc`. Thereby, NetCDF internally utilizes MPI I/O (see handle `MPI-IO #0`) which in turn performs Posix I/O operations (see `POSIX I/O #20`) on `RCE_dump_329.nc`. This view also shows that MPI I/O opens (blue bars) `maps`-files from the `/proc` filesystem through the ISO-C API. Each I/O server process reads (red boxes) its `maps`-file before transferring simulation data to the NetCDF file.

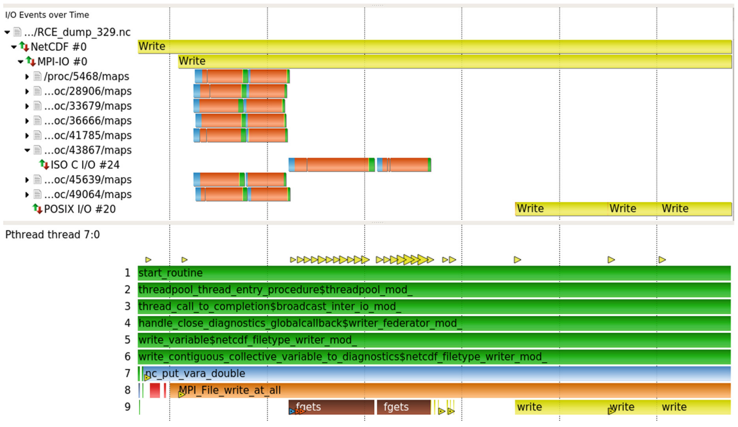

I/O Events over Time
.../RCE_dump_329.nc
NetCDF #0
Write
MPI-IO #0
Write
/proc/5468/maps
...oc/28906/maps
...oc/33679/maps
...oc/36666/maps
...oc/41785/maps
...oc/43867/maps
ISO C I/O #24
...oc/45639/maps
...oc/49064/maps
POSIX I/O #20
Write
Write
Write
Pthread thread 7:0
1 start_routine
2 threadpool_thread_entry_procedure$threadpool_mod_
3 thread_call_to_completion$broadcast_inter_io_mod_
4 handle_close_diagnostics_globalcallback$writer_federator_mod_
5 write_variable$netcdf_filetype_writer_mod_
6 write_contiguous_collective_variable_to_diagnostics$netcdf_filetype_writer_mod_
7 nc_put_vara_double
8 MPI_File_write_at_all
9 fgets fgets write write write

Fig. 7. The I/O timeline (top) shows individual I/O operations of Thread 7 from Rank 0 on specific files. The process summary (bottom) depicts the call stack. (Color figure online)

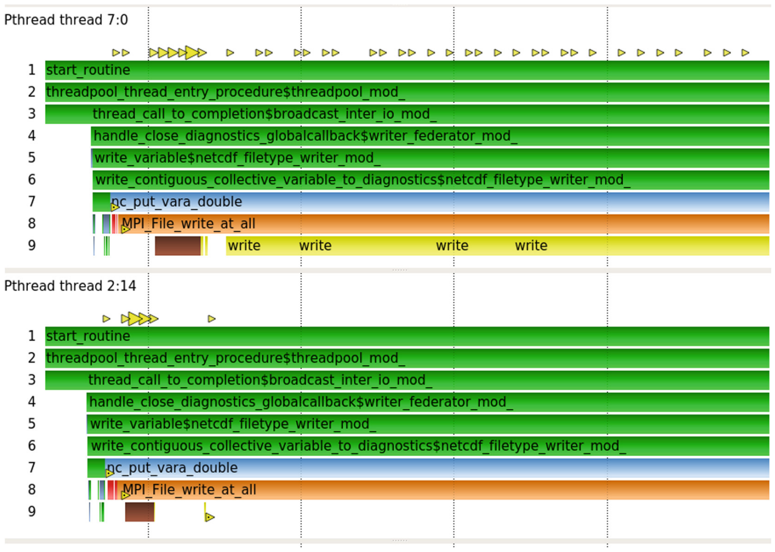
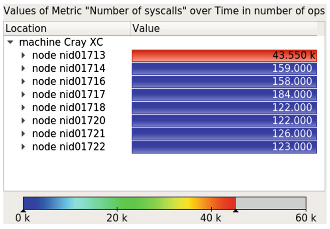

Pthread thread 7:0
1 start_routine
2 threadpool_thread_entry_procedure$threadpool_mod_
3 thread_call_to_completion$broadcast_inter_io_mod_
4 handle_close_diagnostics_globalcallback$writer_federator_mod_
5 write_variable$netcdf_filetype_writer_mod_
6 write_contiguous_collective_variable_to_diagnostics$netcdf_filetype_writer_mod_
7 nc_put_vara_double
8 MPI_File_write_at_all
9 write write write write
Pthread thread 2:14
1 start_routine
2 threadpool_thread_entry_procedure$threadpool_mod_
3 thread_call_to_completion$broadcast_inter_io_mod_
4 handle_close_diagnostics_globalcallback$writer_federator_mod_
5 write_variable$netcdf_filetype_writer_mod_
6 write_contiguous_collective_variable_to_diagnostics$netcdf_filetype_writer_mod_
7 nc_put_vara_double
8 MPI_File_write_at_all
9

Fig. 8. Call stack comparison of two different I/O server processes. (Color figure online)


Values of Metric "Number of syscalls" over Time in number of ops

Location	Value
machine Cray XC	
node nid01713	43.550 k
node nid01714	159.000
node nid01716	158.000
node nid01717	184.000
node nid01718	122.000
node nid01720	122.000
node nid01721	126.000
node nid01722	123.000

Fig. 9. Number of syscalls in MPI I/O mapped to the system tree topology.

The bottom chart in Fig. 7 depicts the process timeline for Thread 7 of Rank 0 and provides details about the calling context of I/O operations in this time slice. For example, the execution of `nc_put_vara_double` (bottom chart, level 7) creates an I/O write event of the `NetCDF #0` handle (top chart). This operation in turn calls `MPI_File_Write_at_all` (bottom chart, level 8) which generates the I/O write event of the `MPI-IO #0` handle (top chart). Level 9 in the bottom chart shows internal details of this collective MPI I/O routine. It depicts the `fgets` call to access the `maps`-file (`/proc/43867/maps`). `write` calls to store the final data which correspond to the write events of the `POSIX I/O #20` handle (top chart). Interestingly, NetCDF executes MPI communication operations (bottom chart, level 8, red bars) within the `nc_put_vara_double` routine. In this time slice, these operations are small compared to the `MPI_File_Write_at_all` routine and do not impede performance. However, in a different scenario, these functions may lead to a communication bottleneck or undesirable wait states.

So far we investigated only one I/O server process. In the next analysis we will compare different I/O server processes. Figure 8 shows the process timelines of I/O server Rank 0/Thread 7 (top) and Rank 14/Thread 2 (bottom). Both servers call identical functions with similar durations until call level 9. On this level, both servers perform ISO-C I/O operations (brown bars) at the beginning of `MPI_File_Write_at_all`. Then, one server process (top) executes `write` functions. It seems that only one I/O server process accesses the `RCE_dump_329.nc` file through the collective I/O operation. The collective operation appears to synchronizes all processes (causing waiting time) except process Rank 0/Thread 7, that performs the actual I/O operations. Figure 9 depicts the number of syscalls within MPI I/O routines aggregated per compute node. Node `nid01713` performs the most syscalls within MPI. This confirms, that only one I/O server transfers data to the `RCE_dump_329.nc` file. Reasons could be the (small) data size or missing support for parallel accesses in the current implementation. For MONC, our analysis suggests optimization potential by switching from collective operations to individual accesses per I/O server process.

6 Conclusions

This work presents a methodology for recording calls to I/O libraries on multiple layers of the software stack. In contrast to current approaches, our methodology explicitly correlates operations between multiple I/O libraries. This enhanced level of detail in the recorded performance data is essential for understanding the overall I/O behavior of applications. Consequently, users can now identify root causes of I/O bottlenecks inside a complex I/O stack. We prove the applicability of our approach in an analysis study of the Met Office/NERC Cloud Model (MONC) code.

7 Future Work

In this work, we show that our approach records valuable information about the I/O behavior of applications. With an intuitive presentation of this information, we support application developers in optimizing I/O-intensive applications. Currently, we are working on integrating our approach into the Score-P open-source measurement infrastructure and OTF2 trace format. Consequently, it will be available in one of the next official Score-P releases. Meanwhile, we provide a prototype implementation [25].

Automatic analysis as a complementary technique to visualization directly guides users to performance bottlenecks. Tools like Scalasca or Casita [21] apply detection mechanisms to identify inefficiency patterns in MPI message transfers or computation offloading to accelerator devices. Similar analysis techniques can be applied to our I/O performance data recordings.

This work focuses on performance analysis of file I/O operations. However, it can be easily extended to monitor I/O operations on sockets. This use case would only require new definitions for representing sockets as an I/O resource (besides files and directories). Furthermore, we plan to add information about failed operations to the current records. This would extend their usability from performance analysis to debugging and correctness checking applications.

Data Availability Statement and Acknowledgments. This research was undertaken as part of the NEXTGenIO project, which is funded through the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 671951. The datasets and code generated during and/or analysed during the current study are available in the figshare repository: https://doi.org/10.6084/m9.figshare.6384164 [25].

References

1. Adhianto, L., et al.: HPCTOOLKIT: tools for performance analysis of optimized parallel programs. In: Concurrency and Computation: Practice and Experience (2010). https://doi.org/10.1002/cpe.1553
2. Knüpfer, A., et al.: The vampir performance analysis tool-set. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68564-7_9
3. Archer Hardware Specification, February 2018. https://www.archer.ac.uk/about-archer/hardware

4. Arm MAP - Low-Overhead Profiling to Optimize C, C++, Fortran and F90 Codes, February 2018. https://www.arm.com/products/development-tools/hpc-tools/cross-platform/forge/map
5. blktrace(8) - Linux man page, February 2018. https://linux.die.net/man/8/blktrace
6. Brendel, R., Wesarg, B., Tschüter, R., Weber, M., Ilsche, T., Oeste, S.: Generic library interception for improved performance measurement and insight. In: Proceedings of the 6th Workshop on Extreme Scale Programming Tools, ESPT 2017, November 2017
7. Brown, N., et al.: A highly scalable met office NERC cloud model. In: Proceedings of the 3rd International Conference on Exascale Applications and Software, EASC 2015, pp. 132–137 (2015)
8. Carns, P., et al.: Understanding and improving computational science storage access through continuous characterization. Trans. Storage **7**(3), 8:1–8:26 (2011). https://doi.org/10.1145/2027066.2027068
9. Chaarawi, M., Gabriel, E., Keller, R., Graham, R.L., Bosilca, G., Dongarra, J.J.: OMPIO: a modular software architecture for MPI I/O. In: Cotronis, Y., Danalis, A., Nikolopoulos, D.S., Dongarra, J. (eds.) EuroMPI 2011. LNCS, vol. 6960, pp. 81–89. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24449-0_11
10. Cyrille Rossant: Should you use HDF5? February 2018. http://cyrille.rossant.net/should-you-use-hdf5/
11. Dieter An Mey and others: Score-P: A Unified Performance Measurement System for Petascale Applications. In: Competence in High Performance Computing (2012)
12. Eschweiler, D., et al.: Open trace format 2 - the next generation of scalable trace formats and support libraries. In: Proceedings of the 14th Biennial ParCo Conference on Applications, Tools and Techniques on the Road to Exascale Computing. Advances in Parallel Computing, vol. 22, pp. 481–490 (2012)
13. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The scalasca performance toolset architecture. Concurr. Comput.: Pract. Exp. **22**(6), 702–719 (2010). https://doi.org/10.1002/cpe.v22:6
14. iostat, February 2018. https://github.com/sysstat/sysstat
15. iotop, February 2018. http://guichaz.free.fr/iotop/
16. Lustre, February 2018. http://lustre.org/
17. Mix, H., Herold, C., Weber, M.: Visualization of multi-layer I/O performance in vampir. In: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2018
18. MPI Forum: MPI: A Message-Passing Interface Standard, Version 3.1, 14 June 2015. https://www.mpi-forum.org/docs/mpi-3.1/. Accessed May 2018
19. Müller, M., et al.: Developing scalable applications with Vampir, VampirServer and VampirTrace. In: Parallel Computing: Architectures, Algorithms and Applications. Advances in Parallel Computing, January 2007
20. sar(1) - Linux man page, February 2018. https://linux.die.net/man/1/sar
21. Schmitt, F., Stolle, J., Dietrich, R.: CASITA: a tool for identifying critical optimization targets in distributed heterogeneous applications. In: 43rd International Conference on Parallel Processing Workshops, pp. 186–195, September 2014. https://doi.org/10.1109/ICPPW.2014.35
22. Shende, S., Malony, A.D., Spear, W., Schuchardt, K.: Characterizing I/O performance using the TAU performance system. In: PARCO, pp. 647–655 (2011)

23. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI-IO portably and with high performance. In: Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems, IOPADS 1999, pp. 23–32 (1999). https://doi.org/10.1145/301816.301826
24. The HDF Group: Hierarchical Data Format, version 5, February 1997–2018. http://www.hdfgroup.org/HDF5/
25. Tschueter, R., Herold, C., Wesarg, B., Weber, M.: Score-P measurement system code and event logs for Euro-Par 2018 paper: a methodology for performance analysis of applications using multi-layer I/O. figshare. Fileset (2018). https://doi.org/10.6084/m9.figshare.6384164
26. Unidata: Network Common Data Form (NetCDF) [software] (2018). https://doi.org/10.5065/D6H70CW6,https://doi.org/10.5065/D6H70CW6
27. Virtual Institute for I/O, February 2018. https://www.vi4io.org/start
28. Vijayakumar, K., Mueller, F., Ma, X., Roth, P.C.: Scalable I/O tracing and analysis. In: Proceedings of the 4th Petascale Data Storage Workshop, PDSW 2009 (2009). https://doi.org/10.1145/1713072.1713080