A Semantics-Aware I/O Interface for High Performance Computing

Michael Kuhn

University of Hamburg
michael.kuhn@informatik.uni-hamburg.de

Abstract. File systems as well as I/O libraries offer interfaces which can be used to interact with them, albeit on different levels of abstraction. While an interface's syntax simply describes the available operations, its semantics determine how these operations behave and which assumptions developers can make about them. There are several different interface standards in existence, some of them dating back decades and having been designed for local file systems. Examples are the POSIX standard for file system interfaces and the MPI-I/O standard for MPI-based I/O.

Most file systems implement a POSIX-compliant interface to improve portability. While the syntactical part of the interface is usually not modified in any way, the semantics are often relaxed to reach maximum performance. However, this can lead to subtly different behavior on different file systems, which in turn can cause application misbehavior that is hard to track down.

On the other hand, providing only fixed semantics also makes it very hard to achieve optimal performance for different use cases. An additional problem is the fact that the underlying file system does not have any information about the semantics offered in higher levels of the I/O stack. While currently available interfaces do not allow application developers to influence the I/O semantics, applications could benefit greatly from the possibility of being able to adapt the I/O semantics at runtime.

The work we present in this paper includes the design of our semantics-aware I/O interface and a prototypical file system developed to support the interface's features. Using the proposed I/O interface, application developers can specify their applications' I/O behavior by providing semantical information. The general goal is an interface where developers can specify *what* operations should do and *how* they should behave – leaving the actual realization and possible optimizations to the underlying file system. Due to the unique requirements of the proposed I/O interface, the file system prototype is designed from scratch. However, it uses suitable existing technologies to keep the implementation overhead low.

The new I/O interface and file system prototype are evaluated using parallel metadata benchmarks. Using a single metadata server, they deliver a sustained performance of up to 50,000 lookup and 20,000 create operations per second, which is comparable to – and in some cases, better than – other well-established parallel distributed file systems.

Keywords: Distributed File Systems, I/O Interfaces, I/O Semantics.

© Springer-Verlag Berlin Heidelberg 2013

1 Introduction

High performance computing is an increasingly important tool for scientific computing. It is used to conduct large-scale computations and simulations of complex systems from basically all branches of the natural and technical sciences, such as meteorology, climatology, particle physics, biology, medicine and computational fluid dynamics. These computations and simulations are usually realized in the form of parallel applications. They use threads, message passing or a combination of both to distribute and speed up the computational work across a supercomputer. Additionally, high performance computing is invaluable in analyzing the large amounts of data produced by such applications.

An important aspect is high performance I/O, because storing and retrieving such large amounts of data can greatly affect the overall performance of these applications. A common access pattern produced by these applications involves many parallel processes, each performing non-overlapping access to a shared file.

File systems provide an abstraction layer between the applications and the actual storage hardware, such that application developers do not have to worry about the organizational layout or technology of the underlying storage hardware. Distributed file systems usually stripe data across several storage devices to improve both storage capacity as well as throughput. Parallel file systems allow multiple clients to access the same data simultaneously. Consequently, most file systems used in high performance computing are parallel distributed file systems. Two of the most widely-used file systems today are Lustre [3] and GPFS [17].

Parallel distributed file systems provide one or more I/O interfaces which can be used to access data within the file system. Additional interfaces are available in the form of libraries. Popular choices include POSIX, MPI-I/O, NetCDF and HDF5. Almost all the I/O interfaces found in high performance computing today offer simple byte- or element-oriented access to data and thus do not have any a priori information about what kind of accesses the applications perform. Even though there are some notable exceptions such as ADIOS or NetCDF, even the more advanced I/O interfaces do not offer support to specify additional semantical information about the applications' behavior and requirements. Due to this lack of knowledge about application behavior, optimizations are often based on heuristic assumptions which may or may not reflect the actual behavior.

While the I/O interface defines which I/O operations are available, the I/O semantics describe and define the behavior of these operations. Usually each I/O interface is accompanied by a set of I/O semantics, tailored to this specific interface. The POSIX I/O semantics are probably the most widely-used semantics, even in high performance computing. However, due to being designed for traditional local file systems, they impose unnecessary restrictions on today's parallel distributed file systems. One of these restrictions are the very strict consistency requirements which can lead to performance bottlenecks in distributed environments.

Performing I/O efficiently is becoming an increasingly important problem. While CPU speed and HDD capacity continue to increase by roughly a factor of 1,000 every 10 years [26,25], the speed of HDDs grows much slower: Early HDDs in 1989 delivered about 0,5 MB/s, while current HDDs manage around 150 MB/s [24]. This corresponds to a 300-fold increase of throughput over the last (almost) 25 years. Even newer technologies such as SSDs only offer throughputs of around 600 MB/s, resulting in a total speedup of 1,200. For comparison, over the same period of time, the computational power increased by a factor of 1,000,000.

There are several ways to compensate for this fact: Increasing the efficiency of I/O, using novel storage technologies or simply buying more storage hardware. The JULEA project aims to increase the efficiency of I/O by providing a new semantics-aware I/O interface which should allow applications to make the most of the available storage hardware. It allows specifying the semantics of I/O operations at runtime and supports batch operations to increase performance. The overall goal is to allow the application developer to specify the desired behavior and leave the actual realization to the I/O system.

This paper is structured as follows: The current state of the art with regards to I/O interfaces and semantics is presented in Section 2. In Section 3, the design of our new semantics-aware I/O interface is elaborated. A preliminary evaluation is given in Section 4. Our design is then compared with other related work in Section 5, followed by a conclusion and some ideas for future work in Section 6.

2 State of the Art

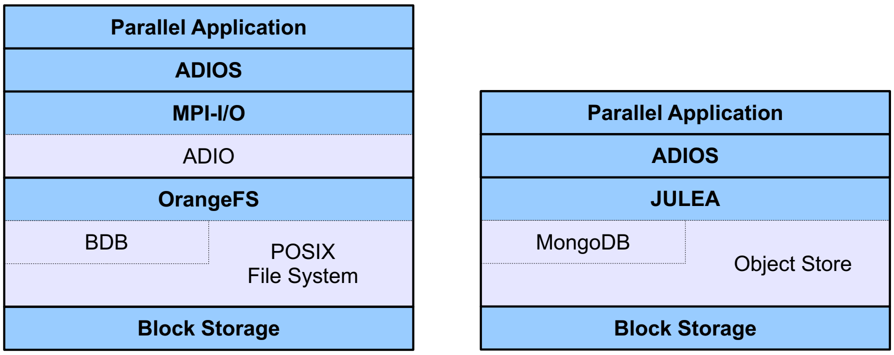
Currently, I/O systems have a strongly layered concept. One major problem with this approach is the fact that the lower layers do not have any information about the upper ones. Due to this, each layer has to perform its own optimizations to be able to use the I/O system's full potential. An example of such an I/O stack can be seen in Figure 1a. The different interfaces such as ADIOS, MPI-I/O and POSIX will be explained below. While the upper layers usually provide more comfort and abstraction, the performance yield might be lower. Therefore, the lower, more difficult-to-use layers are often used directly to harness the I/O system's full potential.

An additional problem is the fact that it is currently not possible to hand semantical information down through the I/O stack. To ease the development of codes in need of high performance I/O it would be very beneficial to provide easy-to-use interfaces that still provide adequate performance.

2.1 I/O Interfaces

Each I/O interface is usually accompanied by its own set of I/O semantics, which are tailored specifically to this interface. A description of the most common I/O interfaces and their corresponding semantics follows.

The **POSIX** I/O interface has been originally designed for use in local file systems. Its first formal specification dates back to 1988, when it was included in POSIX.1. Asynchronous/synchronous I/O was added in POSIX.1b from 1993.



(a) Strongly-layered traditional I/O system (b) JULEA I/O layers

Fig. 1. Comparison of traditional and JULEA I/O stacks

This interface is very widely used, even in parallel distributed file systems, and thus provides excellent portability.

The original interface did not offer ways to specify semantical information about the accesses or the data. A feature added in POSIX.1-2001 is called `posix_fadvise()` and allows announcing the pattern which will be used to access the data. However, this does not change the semantics of any following I/O operations. It typically used to increase the readahead window (`POSIX_FADV_SEQUENTIAL`), disable readahead (`POSIX_FADV_RANDOM`), or to populate (`POSIX_FADV_WILLNEED`) and free (`POSIX_FADV_DONTNEED`) the cache.

The **MPI-I/O** interface was introduced in the MPI-2.0 standard in 1997 [11] and offers support for parallel I/O. It provides an I/O middleware which abstracts from the actual underlying file system – the popular ROMIO implementation uses the so-called ADIO layer which includes support and optimizations for POSIX, NFS, OrangeFS and many others. The MPI-I/O interface uses the existing MPI infrastructure of MPI datatypes to access data within files.

The actual interface looks very much like the POSIX interface using file handles to access files. MPI-I/O offers support for different file access modes, which can be specified at file-open time. The MPI standard specifies several access modes [12]. However, the only to access modes which can be considered semantical information are `MPI_MODE_UNIQUE_OPEN` and `MPI_MODE_SEQUENTIAL` as these give information about *how* the file is going to be accessed.

ADIOS [10] provides a high-level I/O interface that abstracts from the usual byte- or element-oriented access as found in POSIX or MPI-I/O. It outsources the actual I/O configuration into an external XML file which can be used to describe which data structures should be accessed and to automatically generate C or Fortran code. Due to this automatically generated code, the application developer does not need to directly interact with the underlying I/O middleware or file system.

There are a number of other I/O interfaces like SIONlib [5], NetCDF [15] and HDF5 [21] which focus on solving performance problems or offering additional features such as annotated data storage. As they also do not offer semantical information to be specified, they are only mentioned briefly here.

2.2 I/O Semantics

In the following, the most common I/O semantics are presented and potential shortcomings are highlighted.

POSIX I/O features very strict consistency requirements. For example, `write` operations have to be atomic and have to be visible to other clients immediately after the system call returns. While this might be relatively easy to support in local file systems, it can pose a serious bottleneck in parallel distributed file systems, because it effectively prohibits client-side caching from being used and requires additional locking. The semantics can only be changed in a very limited fashion. For example, the `strictatime`, `relatime` and `noatime` options change the file system's behavior regarding the last access timestamp, which can have an impact on performance. Additional options for `async` and `sync` are also available. However, all of these options can only be specified on a per-mount basis and have to be fixed at mount time – that is, they can not be modified by users under normal circumstances.

The **NFS** protocol provides close-to-open cache consistency by default, which implies that changes performed by a client are only written back to the server when the client closes the modified file. However, NFS offers limited support for changing this behavior: By mounting NFS using the `cto` or `nocto` options close-to-open cache coherence semantics can be switched on or off respectively. Additionally, the `async` and `sync` options can be used to modify the behavior of write operations: While `async` causes writes to only be propagated to the server when necessary[1], `sync` will cause system calls to only return when the data has been flushed to the server. Additional mount options are available to modify the caching behavior of attributes and directory entries. However, as in the POSIX case, these options can only be specified at mount time by the administrator.

MPI-I/O's consistency requirements are less strict than those defined by POSIX [19,4]. By default, MPI-I/O guarantees that non-overlapping write operations will be handled correctly and that changes are immediately visible only to the writing process itself. Other processes first have to synchronize their view of the file to see the changes. For use-cases requiring stricter consistency semantics, MPI-I/O offers the so-called *atomic mode*. The atomic mode specifies that changes will be visible to all process within the same communicator instantly. This can be difficult to achieve, because MPI-I/O allows non-contiguous operations and parallel distributed file systems can stripe single write operations over multiple servers [16,8]. However, apart from the configurable atomic mode, MPI-I/O does not offer any other means of changing the semantics. MPI-I/O

[1] Possible reasons include memory pressure and (un)locking, synchronizing or closing a file.

implementations are free to offer so-called *hints*, which are mainly used to control things like buffer sizes and participating processes. However, because hints are optional, different implementations are free to ignore them [20].

3 Design

As previously shown, the interfaces and semantics currently used for distributed file systems are suboptimal because they are either not well-adapted for the requirements and demands found in high performance computing today or do not allow fine-grained semantical information to be specified [14,18,22]. In this paper, we demonstrate a new I/O interface as well as a file system prototype called JULEA. It has been implemented from scratch to be suited specifically for the requirements found in high performance computing.

3.1 Layers

The intended general architecture of the JULEA I/O stack is illustrated in Figure 1b and features less layers than the traditional one in Figure 1a. This allows concentrating all optimizations into a single layer, reducing the implementation and runtime overhead. An important design goal is to remove the duplication of functionality found in the traditional I/O stack. For example, path lookup should only be performed on the uppermost layer. This can be achieved by eliminating the underlying POSIX file systems and using a suitable object store, which allows objects to be accessed directly using a unique ID. JULEA supports multiple storage backends such as existing POSIX file systems as well as object stores. This allows JULEA to always use the best-suited backend while maintaining compatibility with a wide range of software environments. For the metadata part, we decided to use an existing NoSQL database system called MongoDB [1]. The only remaining deficiency is MongoDB's dependency on an underlying POSIX file system, which we are currently investigating.

3.2 File System Namespace

Traditional file systems allow deeply-nested directory structures. To avoid the overhead caused by this, only a restricted, relatively flat hierarchical namespace is supported. While our approach might be unsuited for a general purpose file system, we explicitly focus on specific use-cases that are commonly found in high performance computing.

It is divided into *stores*, *collections*, and *items*. Each store can contain multiple collections which in turn can contain multiple items. Additionally, items feature a very reduced set of metadata. For example, unimportant information like the time of the last access has been omitted. The goal of these changes is to minimize the overhead during normal file system operation. For example, in traditional POSIX file systems, each component of the potentially deeply-nested path has to be checked for each access. This requires reading its associated metadata, checking permissions, etc., which usually happens sequentially. Additionally, in distributed file systems these operations can be very costly if many (relatively small) network messages are involved.

3.3 Interface

The interface has been designed from scratch to offer simplicity of use while still meeting the requirements of high performance and semantics-awareness. Two major features are the ability to specify semantical information and to batch operations.

It is possible to specify additional information equivalent to the coarse-grained statement "this is a checkpoint" or the more fine-grained "this operation requires strict consistency semantics". This allows the file system to tune operations for specific applications by itself. Additionally, it is possible to emulate well-established semantics as well as mixing different semantics within one application.

All accesses to the file systems are done via so-called *batches*. Each batch can consist of multiple operations. For example, multiple items can be created or different offsets within an item can be accessed in one batch. It is also possible to combine different kinds of operations within one batch. For example, one batch might create a collection and several items within it, and write data to each one. Because the file system has knowledge about all operations within one batch, more elaborate optimizations can be performed. The advantages of this approach will be evaluated in Section 4.

The pseudo code found in Listing 1.1 shows an example of how the interface generally works. A new batch using the POSIX semantics (line 1) as well as a store, collection and item are created (lines 2–4). Afterwards, the collection is added to the store (line 6) and the item is added to the collection (line 7). Additionally, some data is written to the item (line 8). Finally, the batch is executed (line 10) which in turn executes all three operations with the given semantics.

Listing 1.1. JULEA pseudo code

```
 1  batch = new Batch(POSIX_SEMANTICS);
 2  store = new Store("test");
 3  collection = new Collection("test");
 4  item = new Item("test");
 5
 6  batch.add(store.add(collection));
 7  batch.add(collection.add(item));
 8  batch.add(item.write(...));
 9
10  batch.execute();
```

3.4 Semantics

The JULEA interface allows many aspects of the semantics of file system operations to be changed at runtime on a per-batch basis. Several key areas of the semantics have been identified as important to provide opportunities for optimizations. Support for atomicity, concurrency, consistency, persistency and safety has already been designed, while possible data manipulation and security aspects are still in the planning stage.

Other ideas include prompting the file system to store multiple copies of file data and metadata, and to compress or encrypt it on-the-fly. The security policy could be changed depending on the file system environment, enabling or disabling more strict permission checks.

Detailed information about the different semantics together with possible configurations is given in the following list.

- **Atomicity:** The atomicity semantics can be used to specify whether accesses should be atomic, that is, whether it is possible for clients to see intermediate states of operations. For example, a single write operation spanning two servers might have already reached one of them but not the other. If atomic accesses are enabled, other clients will be unable to see this inconsistent state.
- **Concurrency:** The concurrency semantics can be used to specify whether concurrent accesses will take place and, if so, how they will look like. This can be used to enable or disable locking as needed.
- **Consistency:** The consistency semantics can be used to specify if and when clients will see modifications performed by other clients. This can be used to enable client-side read caching whenever possible.
- **Persistency:** The persistency semantics can be used to specify if and when data must be written to persistent storage. This can be used to enable client-side write caching whenever possible. For example, temporary data can be cached more aggressively and does not necessarily need to be written to persistent storage at all. This can be especially advantageous when different levels of storage such as node-local SSDs are available.
- **Safety:** The safety semantics can be used to specify how safely data should be handled. For example, this can be used to disable waiting for the server's acknowledgment when sending unimportant data. On the other hand, it can be used to make sure that important data will survive a system failure by flushing it to the storage devices immediately.
- **Templates:** Semantics templates can be used to provide templates for specific use-cases. For example, it can be used to to mimic the current POSIX semantics as closely as possible, and to tune the semantics for application input or output, which can be handled differently.

3.5 Architecture

JULEA provides a user-space library which can be linked to applications, allowing them to use the JULEA I/O interface. Additionally, a user-space daemon handles storing the file data on the I/O servers. The library communicates with both the JULEA daemons and the MongoDB servers running on the data and metadata servers, respectively. By providing all functionality in user-space, JULEA is largely independent of the used operating system and kernel, and can be easily ported to new software environments.

Operations within one batch are aggregated and sent to the appropriate daemons in as few messages as possible to decrease network overhead. There are also plans to reorder and merge operations, which can help to further increase efficiency. The semantics specified for each batch are used to internally modify the behavior of I/O operations. While most of this semantical information is only needed by the client, it can also be transferred to the daemons when necessary.

For example, the safety semantics are always sent to the daemons, which can use this information to avoid sending back unneeded replies to the clients. The atomicity and concurrency semantics can be used to decide whether locking is necessary. Traditional interfaces do not have access to such information and therefore have to make pessimistic assumptions, which might force them to always handle the worst-case scenario. Since application developers know the access patterns of their applications, they can easily specify such information. This can be very beneficial, because lockless access to shared files can improve performance dramatically. The additional semantical information can also be used to reduce locking overhead on the metadata servers by making sure that specific metadata such as the file size and modification time are only stored explicitly for non-parallel workloads. Since highly parallel workloads would cause metadata update storms, such information is better computed on-the-fly whenever it is needed in these cases.

4 Evaluation

Our prototype was built to provide a reference implementation of our new I/O interface. It has built-in support for tracing client and server activities in various formats such as OTF [7] and HDTrace [13]. This can be used to visualize the inner workings and can be very helpful when debugging errors or searching for performance issues. To evaluate the benefits of our implementation we have extended the fileop benchmark – which is part of IOzone [27] – to support MPI and to also use the native interfaces of OrangeFS and JULEA. Only the most interesting metadata-heavy operations (`mkdir`, `rmdir`, `create`, `stat` and `delete`) were benchmarked. All results are averaged over at least five runs.

Figure 2 shows results for Lustre, OrangeFS and JULEA[2]. fileop was configured to run with a varying number of MPI processes on 1–5 nodes with up to 12 processes per node, each process working in its own directory/collection. OrangeFS and JULEA were tested using their native interfaces, while the POSIX interface was used for Lustre. All file systems were configured to provide one data and metadata server, and used their default configuration – apart from OrangeFS, where `TroveSyncMeta` was set to `no` to disable synchronous metadata operations. OrangeFS and JULEA were run on dual-socket machines with two

[2] Note the logarithmic y-axis and different scaling for each of the subfigures.

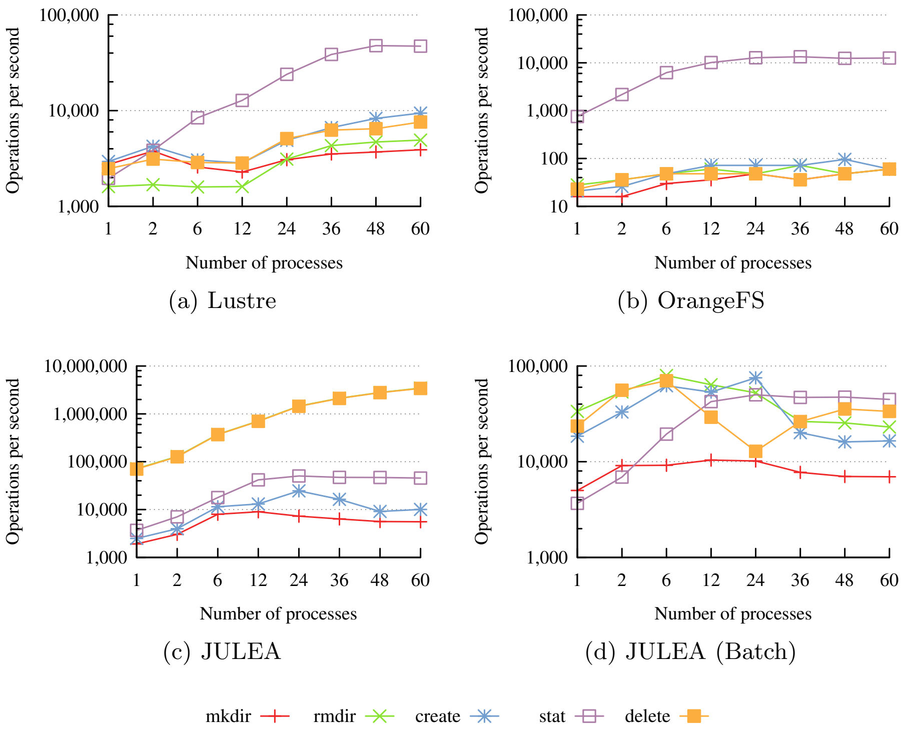


Fig. 2. Comparison of Lustre, OrangeFS and JULEA metadata performance

Intel Xeon X5650 processors and 12 GB of main memory each. Due to different operating system requirements, Lustre was run on single-socket machines with one Intel Xeon E31275 processor and 16 GB of main memory each. The evaluation was carried out using the `ldiskfs` backend for Lustre, an underlying `ext4` file system for OrangeFS and JULEA, and MongoDB 2.2.2 for JULEA. All data was stored on 7,200 RPM HDDs.

The results for Luste are shown in Figure 2a. It is interesting to note that the performance of all operations except for `stat` does not improve if more than 1 or 2 client processes per node are used. For these operations, there is practically no performance difference for the configurations using 1–12 processes. However, performance does increase when processes are distributed across more client nodes. Using 60 client processes, the `mkdir` and `rmdir` operations reach a maximum of around 4,000 and 5,000 operations per second, while the `create` and `delete` operations reach approximately 9,500 and 7,500 operations per second. `stat` scales well up to 36–48 client processes, where the curve begins to flatten, reaching a maximum of roughly 48,000 operations per second.

Figure 2b shows the results for OrangeFS. The `mkdir`, `rmdir`, `create` and `delete` operations deliver 20–30 operations per second when using a single client process. Increasing the number of processes does not significantly improve the results, resulting in a maximum performance of 75–100 operations per second when using 60 client processes. The `stat` operation performs significantly better with 750 operations per second when using a single client process. It also scales well up to 12 processes, where the curve flattens, reaching a maximum of 13,000 operations per second when using 24–60 processes.

The results for JULEA using individual operations is shown in Figure 2c. The `mkdir` and `create` operations deliver 2,000 and 2,500 operations per second using only a single client process. They scale well up to 12 and 24 processes respectively, where they reach their maximum performance of 9,000 and 25,000 operations per second. Increasing the number of processes further causes the performance to drop to around 50%. The `stat` operation starts out with 3,500 operations per second, reaches its maximum of 50,000 operations per second when using 24 processes and stays at this level when further increasing the number of processes. The `rmdir` and `delete` operations show suspiciously high performance of up to 3,400,000 operations per second, which might be caused by the MongoDB query returning before the actual remove operation has finished. However, this initial assumption still has to be investigated.

Figure 2d shows the results for JULEA using batches to aggregate operations. The `mkdir` operation provides only slightly better performance than in the previous case, peaking at 10,000 operations per second. However, fewer client processes are required to reach maximum performance, with only two processes needed to obtain 9,000 operations per second. The `create` operation performs significantly better, starting at 18,500 operations per second and reaching a maximum of 75,000 operations per second with 24 client processes. While performance drops to 20–25% when using 36 processes and more, it is still faster by a factor of 2 when compared to using individual operations. `status` behaves exactly the same as in the previous case. The `rmdir` and `delete` operations now deliver more realistic performance numbers. However, both operations reach a maximum with 6 client processes, dropping steadily after that. As already noted, both operations still have to be investigated more closely. Especially the `delete` operation's steep drop from 6–24 processes warrants a closer examination.

Overall, JULEA provides metadata performance comparable to other established parallel distributed file systems. More investigation and tweaking of the MongoDB configuration will be required to eliminate the performance drop-off with larger amounts of client processes. We are currently also working on supporting sharded configurations of MongoDB which we expect to increase performance even further.

5 Related Work

MosaStore [2] is a versatile storage system which is configurable at application deployment time and thus allows application-specific optimizations. This approach is similar to the JULEA approach, however, MosaStore provides a storage system bound to specific applications instead of a globally shared one. Additionally, the storage system can not be reconfigured at runtime and keeps the traditional POSIX I/O interface.

The authors present a configurable security approach in [6] which allows using scavenged storage systems consisting of unused workstation hardware in trusted, partially trusted and untrusted environments in a secure way. While JULEA does not use scavenged storage hardware, the cited work shows that configurable security can be achieved with relatively low overhead. This could also be supported in JULEA to cater to different security requirements.

A new file system approach is presented in [9] that eliminates the current need for many small accesses to get the metadata of all path components during path lookup. By using the hashed file path to directly look up the related data and metadata, this can be reduced to only require one read operation per file access. While this can significantly increase small file performance, renaming of parent directories causes all child hashes to change which might lead to a lot of computational overhead. The JULEA interface does not use hashed path lookups for this reason, but implements a relatively flat namespace to reduce lookup overhead.

CAPFS [23] introduces a new content-addressable file store that allows users to define data consistency semantics at runtime. While providing a client-side plug-in API allows users to implement their own consistency policies, CAPFS is limited to tuning the consistency of file data and keeps the traditional POSIX interface. Additionally, the consistency semantics can only be changed on a per-file basis.

6 Conclusions and Future Work

In this paper, we have presented the design and implementation of our novel semantics-aware I/O interface and prototypical file system. It provides an interface and semantics suited for high performance computing, and aims to reduce redundant functionalities currently found within the I/O stack. Unlike similar approaches, JULEA allows fine-grained specification of the I/O semantics required by the application on a per-operation basis. By merely specifying the I/O requirements and leaving the realization and potential optimizations to the underlying file system, the I/O system can be tailored better to the actual hardware, improving efficiency. Additional features like the batching of operations allow further optimizations and provide potential for experimenting with other novel ideas regarding the handling of I/O.

One such idea, which fits naturally into the concept of batches, but is more oriented towards programming efficiency, is to implement transaction support. This would allow the application developer to specify how errors should be handled. For example, the file system could be told to automatically revert to the previous state in case of an error, making error handling and subsequent cleanup tremendously more easy.

Additional evaluations focusing on other aspects of the file system such as data performance and the influence of different semantics on a number of use-cases have already been planned and will follow soon. Early benchmarks suggest that shared file access can especially benefit from being able to specify semantical information about the access patterns. In some cases, the strict atomicity and consistency requirements enforced by the traditional POSIX semantics can cause performance drops of a factor of 100 and more. To evaluate the potential benefits with realistic use-cases we also plan to port an existing numerical application to the JULEA I/O interface in the future.

References

1. 10gen, Inc.: MongoDB (2012), `http://www.mongodb.org/` (last accessed: February 2013)
2. Al-Kiswany, S., Gharaibeh, A., Ripeanu, M.: The Case for a Versatile Storage System. SIGOPS Oper. Syst. Rev. (January 2010)
3. Cluster File Systems, Inc.: Lustre: A Scalable, High-Performance File System (November 2002), `http://www.cse.buffalo.edu/faculty/tkosar/cse710/papers/lustre-whitepaper.pdf` (last accessed: February 2013)
4. Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, B., Prost, J.P., Snir, M., Traversat, B., Wong, P.: Overview of the MPI-IO Parallel I/O Interface. In: IPPS 1995 Workshop on Input/Output in Parallel and Distributed Systems (April 1995)
5. Frings, W., Wolf, F., Petkov, V.: Scalable massively parallel I/O to task-local files. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009 (2009)
6. Gharaibeh, A., Al-Kiswany, S., Ripeanu, M.: Configurable security for scavenged storage systems. In: Proceedings of the 4th ACM International Workshop on Storage Security and Survivability, Storage (2008)
7. Knüpfer, A., Brendel, R., Brunst, H., Mix, H., Nagel, W.E.: Introducing the Open Trace Format (OTF). In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3992, pp. 526–533. Springer, Heidelberg (2006), `http://dx.doi.org/10.1007/11758525_71`
8. Latham, R., Ross, R., Thakur, R.: Implementing MPI-IO Atomic Mode and Shared File Pointers Using MPI One-Sided Communication. Int. J. High Perform. Comput. Appl. (May 2007)
9. Lensing, P., Meister, D., Brinkmann, A.: hashFS: Applying Hashing to Optimize File Systems for Small File Reads. In: Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os, SNAPI 2010 (2010)
10. Lofstead, J.F., Klasky, S., Schwan, K., Podhorszki, N., Jin, C.: Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In: Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments, CLADE 2008 (June 2008)
11. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. Version 3.0 (September 2012), `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf` (last accessed: February 2013)
12. Message Passing Interface Forum: Opening a File (February 2013), `http://www.mpi-forum.org/docs/mpi22-report/node265.htm` (last accessed: February 2013)

13. Minartz, T., Molka, D., Kunkel, J., Knobloch, M., Kuhn, M., Ludwig, T.: Tool Environments to Measure Power Consumption and Computational Performance, ch. 31. Chapman and Hall/CRC Press Taylor and Francis Group (2012)
14. Patil, S., Gibson, G.A., Ganger, G.R., Lopez, J., Polte, M., Tantisiroj, W., Xiao, L.: In search of an API for scalable file systems: Under the table or above it? In: Proceedings of the 2009 Conference on Hot Topics in Cloud Computing, HotCloud 2009 (2009)
15. Rew, R., Davis, G.: Data Management: NetCDF: an Interface for Scientific Data Access. IEEE Comput. Graph. Appl. (July 1990)
16. Ross, R., Latham, R., Gropp, W., Thakur, R., Toonen, B.: Implementing MPI-IO atomic mode without file system support. In: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid, CCGRID 2005 (2005)
17. Schmuck, F., Haskin, R.: GPFS: A Shared-Disk File System for Large Computing Clusters. In: Proceedings of the 1st USENIX Conference on File and Storage Technologies, FAST 2002 (2002)
18. Sehrish, S.: Improving Performance and Programmer Productivity for I/O-Intensive High Performance Computing Applications. Phd thesis, School of Electrical Engineering and Computer Science in the College of Engineering and Computer Science at the University of Central Florida (2010)
19. Sterling, T., Lusk, E., Gropp, W. (eds.): Beowulf Cluster Computing with Linux, 2nd edn. MIT Press (2003)
20. Thakur, R., Ross, R., Lusk, E., Gropp, W., Latham, R.: Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation (April 2010), `http://www.mcs.anl.gov/research/projects/romio/doc/users-guide.pdf` (last accessed: February 2013)
21. The HDF Group: Hierarchical data format version 5 (2000-2010), `http://www.hdfgroup.org/HDF5` (last accessed: February 2013)
22. Vilayannur, M., Lang, S., Ross, R., Klundt, R., Ward, L.: Extending the POSIX I/O Interface: A Parallel File System Perspective. Tech. Rep. ANL/MCS-TM-302 (October 2008)
23. Vilayannur, M., Nath, P., Sivasubramaniam, A.: Providing Tunable Consistency for a Parallel File Store. In: Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies, FAST 2005, vol. 4 (2005)
24. Wikipedia: Festplattenlaufwerk – Geschwindigkeit (February 2013), `http://de.wikipedia.org/wiki/Festplattenlaufwerk#Geschwindigkeit` (last accessed: February 2013)
25. Wikipedia: Mark Kryder – Kryder's Law (February 2013), `http://en.wikipedia.org/wiki/Mark_Kryder#Kryder.27s_Law` (last accessed: February 2013)
26. Wikipedia: TOP500 (February 2013), `http://en.wikipedia.org/wiki/TOP500` (last accessed: February 2013)
27. Norcott, W.D., Capps, D.: IOzone Filesystem Benchmark (2006), `http://www.iozone.org/` (last accessed: February 2013)