DOI: 10.14529/jsfi190203

How File-access Patterns Influence the Degree of I/O Interference between Cluster Applications

Aamer Shah[1], *Chih-Song Kuo*[2], *Akihiro Nomura*[3], *Satoshi Matsuoka*[4], *Felix Wolf*[1]

On large-scale clusters, tens to hundreds of applications can simultaneously access a parallel file system, leading to contention and, in its wake, to degraded application performance. In this article, we analyze the influence of file-access patterns on the degree of interference. As it is by experience most intrusive, we focus our attention on write-write contention. We observe considerable differences among the interference potentials of several typical write patterns. In particular, we found that if one parallel program writes large output files while another one writes small checkpointing files, then the latter is slowed down when the checkpointing files are small enough and the former is vice versa. Moreover, applications with a few processes writing large output files already can significantly hinder applications with many processes from checkpointing small files. Such effects can seriously impact the runtime of real applications—up to a factor of five in one instance. Our insights and measurement techniques offer an opportunity to automatically classify the interference potential between applications and to adjust scheduling decisions accordingly.

Keywords: performance, I/O, file-access pattern, interference, benchmarking.

Introduction

The computational demand of HPC applications is continuously growing, raising the performance expectations of cluster users to unprecedented levels. In order to accommodate such demands, HPC systems frequently employ specialized designs such as multi-dimensional torus networks, GPU-based accelerators, and powerful parallel file systems. The latter are needed to provide service for an enormous amount of file accesses in parallel. Such parallel file systems are installed as centralized resources with a middle layer of I/O servers connected to storage devices at one end and to compute nodes at the other. Decoupling compute resources from I/O resources allows for better management and scalability of the I/O subsystem. However, the centralized design also means that multiple applications may share the same file system. This can lead to contention in the event of simultaneous file access and can substantially degrade application performance. Applications that perform frequent file access requests or access massive amounts of data are especially sensitive to such conditions, adding an element of variability to their performance [36].

HPC applications that perform frequent or massive file access requests are quite common. Examples include data-intensive codes such as MADCAP cosmic microwave background analyzer [34] and GCRM global cloud system resolving model [40]. They both write massive amounts of data during execution, resulting in numerous write requests. In contrast, OpenFOAM continuum mechanics solver [17] and Community Atmosphere Model (CAM) [22] of the Community Earth System Model (CESM) [33] frequently checkpoint their state, resulting in small but recurring writes. Overall, very different classes of file-access patterns can be distinguished. Not only do these patterns access the file system in unique ways, but their sensitivity to interference

[1]Laboratory for Parallel Programming, Technische Universität Darmstadt, Darmstadt, Germany
[2]Taiwan Semiconductor Manufacturing Company Limited, Hsinchu, Taiwan
[3]Global Scientific Information and Computing Center, Tokyo Institute of Technology, Tokyo, Japan
[4]Department of Mathematical and Computing Science, Tokyo Institute of Technology, Tokyo, Japan

from other applications that access the file system at the same time also varies widely. Likewise, they actively interfere with other I/O-intensive applications in different ways. All this makes access patterns to be an important factor for file-system contention. Our initial experiments with different access patterns revealed negligible interference in the case of read-read and read-write contention. These results are also consistent with word-of-the-mouth understanding in the HPC community. Therefore, we concentrated our investigation on write-write contention and the most common access patterns involved.

File system contention and the associated performance degradation are well-known [8]. In this context, the influence of request size and process count has already been studied from a single-application perspective [21], while process count has been identified as a factor of dominance when two applications compete for the file system [29]. Similarly, file-access patterns have been studied in various contexts [12, 14, 20, 25, 37]. The novelty of our research is that we study common write patterns found in HPC applications from the perspective of simultaneous access from different applications. To this end, we first developed a micro-benchmark capable of producing three distinct file-access patterns, simulating those of real applications. Two of these patterns mimic application checkpointing and out-of-core processing, while the third pattern mimics writing large files. We explored the interference potential of these patterns by running them simultaneously against each other, in the form of either micro-benchmarks or realistic applications covering check-point-intensive and data-intensive access patterns. We not only observed different levels of interference between different patterns, but also identified some general rules such as writing large output files dominating checkpointing at smaller checkpoint sizes, with the trend being reversed for larger checkpoint sizes.

In our previous work, we analyzed write access patterns and their effect on interference [6]. In this work, we expand on the topic with a more realistic checkpointing pattern, evaluate how the interference potential depends on the number of processes the application runs with, and confirm our findings with a larger set of production codes. We summarize our contributions as follows:

- An experiment design that allows the quantification of interference between different file-access patterns.
- An I/O-server monitoring capability added to the hitherto purely application-centric interference profiler LWM^2 [9], enabling us to isolate distinct interference phenomena even in noisy environments.
- An analysis of the interference potential of common file write patterns in HPC applications, including the identification of a typical combination with high interference potential.

Taken together, our results pave the way for an effective reduction of interference in the future. Specifically, it brings us much closer to the automatic recognition of applications with high interference potential, allowing their I/O to be separated either in space or time.

The remainder of the paper is organized as follows. First, we provide the necessary background information on parallel file systems in Section 1. In Section 2, we present our approach, including a taxonomy of file-access patterns, an explanation of our experiment design, an introduction to the interference profiler LWM^2, and a description of the I/O server monitor added to LWM^2 for the purpose of this study. After that, we present our results in Section 3, ranging from micro-benchmark-only experiments to measurements with realistic applications. Finally, we review related work in Section 4 before we draw our conclusions and outline future perspectives in Conclusion section.

1. Parallel File Systems

In order to accommodate an increasing number of concurrent file accesses, cluster file systems evolved from a simple client-server model in the style of NFS into usually dedicated clusters of servers and storage devices called parallel file systems. In the most common configuration, a parallel file system connects servers and storage devices via a dedicated network, while it connects servers to compute nodes via a shared message-passing network, as shown in Fig. 1. Clients running on compute nodes forward file-access requests to the I/O servers. Then I/O servers then distribute them to the attached storage devices—according to the mapping of files onto storage devices. This allows handling simultaneous file accesses with better performance. Additionally, striping individual files across multiple storage devices supports efficient parallel access to a single file. Following these general design principles, several implementations such as Lustre, GPFS, FhGPS, PVFS, PanFS, and HDFS emerged. Below, we describe two popular parallel file systems used in our experiments in more details.

1.1. Lustre

Lustre is a file-storage system for clusters used by many of the Top500 HPC systems [24]. It offers up to petabytes of storage capacity and provides multiple gigabytes per second of I/O throughput. Its architecture distinguishes two basic types of servers: *metadata servers* (MDSs) and *object storage servers* (OSSs), as shown in Fig. 1. An MDS stores file-system structure information, including directory layout and file attributes. An OSS stores the actual file-data stripes on the attached *object storage targets* (OSTs). When an I/O request is made, MDS and OSS internally perform different types of file accesses. The MDS performs search and small read and write operations on the file structure information, while the OSS performs potentially large reads and writes on the actual file. Decoupling metadata from data makes it possible to optimize each server type for its most frequent access pattern.

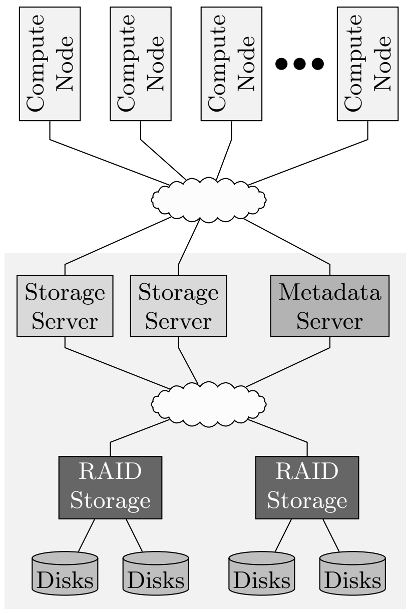


Figure 1. A typical Lustre configuration, with separate I/O servers for metadata and file storage

1.2. GPFS

General Parallel File System (GPFS) is a proprietary parallel file system developed by IBM [3]. It is often found on Blue Gene systems but is also available on other HPC clusters such as TSUBAME 2.5. It supports multiple configurations, including the shared-disk-cluster configuration, in which every compute node manages a part of the file system. However, on large HPC systems, a separate I/O subsystem is more common. In such a configuration, GPFS can span an I/O subsystem with thousands of nodes. GPFS stores data files and their associated metadata on the same block-based devices called *network shared disks* (NSDs). This makes GPFS also suitable for applications with small file accesses such as Web servers. GPFS stripes data files across all disks in a storage pool, achieving high performance. In addition, internal storage pools can be defined to provide different levels of availability and performance for certain files.

2. Approach

Many HPC applications are data-intensive, that is, they perform extensive I/O operations. They employ different I/O libraries and file formats and produce different process-to-file ratios. Because of the fact that a significant proportion of applications still use POSIX-IO or MPI-IO in the classic one-file-per-process manner [15], we concentrate our experiments on this configuration, while also evaluating MPI shared-file scenarios. Given that using MPI-IO with one file per process is essentially equivalent to POSIX-IO [34], at least on our test systems and on many others, our micro-benchmarks exercise only MPI-IO.

Algorithm 1 Open-Write-Close

```
loop
    Open New File
    Write $chunksize$
    Flush I/O Writes
    Close File
end loop
```

Algorithm 2 Write-Seek

```
Open File
loop
    Seek to the beginning
    Write $chunksize$
    Flush I/O Writes
end loop
Close File
```

Algorithm 3 Aggregate-Write

```
Open File
loop
    Write $chunksize$
end loop
Close File
```

Figure 2. Three I/O access patterns

2.1. File Access Patterns

HPC applications exhibit a variety of file access patterns, whose frequent checkpointing, file accesses for out-of-core processing, and writing of large output files are considered here. We implemented three characteristic patterns corresponding to these three use cases as micro-benchmarks, ran them with a range of file sizes, and measured their interference potential when executed against each other as well as against realistic applications. Figure 2 shows the pseudo-code of the three patterns.

Open-write-close. The first considered pattern we consider is called open-write-close (OWC) (Listing 1). In this pattern, each process creates a new file, writes data to it and then closes it. In the next iteration, a new file is created again for data writing. The pattern is commonly used for checkpointing in many applications, such as Flash [30], CESM [33] and OpenFOAM [17]. This access pattern generates a large number of metadata operations, while the actual amount of data written to files can be small. On systems with limited metadata resources such patterns can quickly create a bottleneck at scale. Compared to our previous work [6], we have updated the open-write-close pattern to create a new file in each iteration, mimicking the checkpointing pattern in a more realistic fashion.

Write-seek. In the write-seek (WS) pattern (Listing 2), a process opens a file at the beginning. It then writes a chunk of data to it, and then seeks back to the beginning of the file. At the end of execution, the process closes the file. This pattern is similar to the open-write-close pattern in the sense that it performs a massive number of small file accesses. However, it generates less metadata traffic as it reuses the same file, keeping it continuously open. Between individual writes, only seek operations take place. The write-seek pattern captures a simplified version of file accesses during out-of-core processing of HPC applications, such as in MADCAP [28]. Facing memory capacity pressure, HPC applications often have to resort to out-of-core processing. This means they write data they cannot hold in the main memory temporarily to a file, and read it back once it needs to be processed. This results in a write-seek-read pattern. The pattern can have many different instantiations with respect to write size, seek size, and read size. For simplicity as our goal is measuring write-write interference potential, we have reduced the pattern to a write followed by a complete seek.

Aggregate-write. In the aggregate-write (AW) pattern (Listing 3), a process opens a file at the beginning and then continues to append chunks of data to it. The file gets closed at the end of execution. This pattern is similar to large writes in such applications as MADCAP [34] and GCRM [40]. The pattern involves a few metadata operations but many write operations, resulting in large file sizes. At scale, this pattern can substantially challenge the performance of an I/O subsystem.

Client-side I/O caching requires flushing the I/O traffic after every write operation for the open-write-close and the write-seek patterns. Otherwise, writes of small chunks remain cached in buffers for each OST in the Lustre client and are overwritten with the next write. We have also found flushing of write buffers in real applications to be a common practice. Therefore, our addition of buffer flushes is not unusual. The need for flushes does not arise for writes of large chunks if the chunk size is larger than the OST buffer size. Moreover, this issue does not affect aggregate-write, in which small writes are initially collected in the OST buffer and eventually are committed to the file system. In order to have a consistent benchmark, writes were flushed for both Lustre and GPFS, and for all chunk sizes.

2.2. Capturing Interference

To capture incidents of interference, we run the patterns side by side and measure the change in throughput in comparison to an isolated run. We call the benchmark whose throughput degradation we are interested in the *probe*. The throughput degradation serves as a quantification of the *passive* interference it suffers. The benchmark causing this degradation through *active* interference is called the *signal*. To study the way that interference effects evolve over the runtime of a specific, more complex probe application, we let the signal benchmark to also produce its pattern in a *periodic* fashion, with I/O activity being interrupted by silent phases without the I/O activity. Whenever the signal shows activity, the probe may suffer a dent whose depth indicates the severity of the interference.

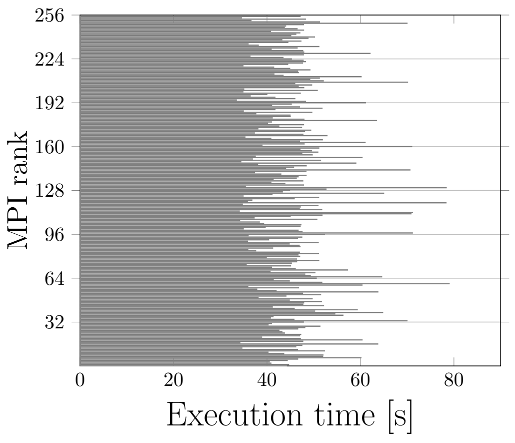
In order to measure how the I/O throughput of an application changes, we use the profiler LWM^2 [9] after extending it to suit our requirements. LWM^2 is a lightweight profiler designed to collect the most basic performance metrics with as little overhead as possible. The I/O metrics relevant to our study are all measured in dynamically loaded interposition wrappers. One aspect important to our study is the ability of LWM^2 to represent performance dynamics in *time slices*. In addition to production of a compact performance summary covering the entire runtime, LWM^2 splits the execution into fixed-length time slices and generates a profile for each of them. The time slice boundaries are synchronized across the entire system by aligning them with the system time. As a result, the simultaneity of performance phenomena occurring in different applications can be easily established. This is useful because it may indicate a causal relationship between these phenomena. The duration of time slices is configurable. In our experiments, we use a time-slice length of 4 seconds and a period length of 24 seconds for the periodic version of our micro-benchmarks. In this way, each period covers at least a few time slices.

However, the mostly application-centric perspective of LWM^2 confronts us with two challenges: noise from other applications not related to our experiments and irregular behaviors of the I/O servers themselves. Ideally, I/O interference experiments should be conducted in a fully controlled, noise-free environment. In practice, however, reserving an entire production cluster for an extended period of time is too expensive. Moreover, the throughput delivered by I/O servers is often non-uniform. For example, the exhaustion of cache space may result in a sudden throughput drop. As a consequence, such irregularities may further blur the interference effects we want to study.

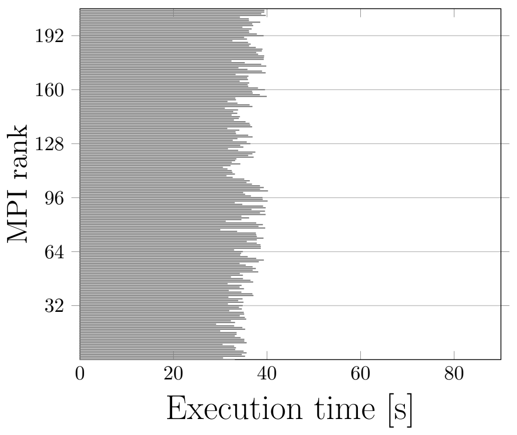
In order to be able to keep our measurements as clean as possible from these two effects, we extended LWM^2 to monitor activities of the I/O server during execution of an application as well. The server activities are captured in every time slice, allowing us to correlate events across applications and I/O servers. In particular, this allows runs to be filtered out where the file-server load is 10% higher than the application I/O traffic captured by POSIX/MPI-IO wrappers. In addition, server-side monitoring allowed us to learn more about certain non-uniform but to some degree predicable behaviors, which we are now able to exclude from our measurements, as explained in Section 2.3. For both GPFS and Lustre, we estimated the I/O traffic to and from the servers by profiling the InfiniBand counters of the servers. Moreover, for Lustre, we parsed the diagnostic data updated by the Lustre client software running on each node to capture the amount of reads and writes from/to the I/O servers.

2.3. Server-side Imbalance

In some experiments, we observed substantial differences among the execution times of individual processes of an application that occurred sporadically with both file systems. In such cases, most processes finished within the expected time, while the remaining ones had to keep performing I/O for a significantly longer duration, sometimes more than twice as long, as shown in Fig. 3a. Such observations are not uncommon and have been reported before [5]. One major factor revealed in a closer investigation of the imbalance effect was unbalanced load on the file-



a) Default



b) 1 file / OST

Figure 3. Mapping one file to one OST reduces the runtime imbalance among processes

server side, as shown in Fig. 4. In particular, this happened with Lustre, where files are randomly assigned to an OST in the one-file-per-process mode.

When an OST was shared by many processes, its performance dropped, which in turn affected the throughput of the associated I/O server. We confirmed this observation by artificially enforcing an equal number of files per OST in a small experimental run, which reduced the disparity of execution time by more than 75%, as shown in Fig. 3b. However, such enforcement is not feasible in a real world scenario, as it requires the number of processes to be a multiple of the number of OSTs. With GPFS, the process imbalance effect occurred to a lesser extent with large files because they were automatically striped across all NSDs, but more predominantly with small files below the stripe size presumably for the opposite reason. Besides the OST/NSD load imbalance, other factors, such as the straggler phenomenon [5], might also contribute to the imbalance.

To accommodate the variance resulting from this imbalance, while still being able to discern interference effects, we considered only the balanced part of a run. This approach is justifiable since the imbalance only affects the later stage of a run, in which only a small portion of the total I/O volume is written. In practice, we found that the I/O traffic in this tail-off stage is usually less than 10%. As a result, we calculated the throughput drop and runtime dilation, our comparison metrics, only up to the moment when the first of the two simultaneously running programs had written 90% of its data volume. Even though this empirical technique did not completely remove the effects of the server-side imbalance, it reduced the resulting imprecision significantly and consistently, while preserving the effect of interference.

3. Evaluation

This section presents the results of our interference experiments. In these experiments, we first ran pairs of our micro-benchmarks against each other to study the interaction of the different patterns in their purest form. To confirm our findings, we then executed the micro-benchmarks against three realistic applications, OpenFOAM, MADbench2, and HACCIO, used for simulations of fluid dynamics, cosmic background radiation, and collisionless cosmic fluid creation, respectively. Finally, we analyzed the interference effect observable between two instances of each of these applications.

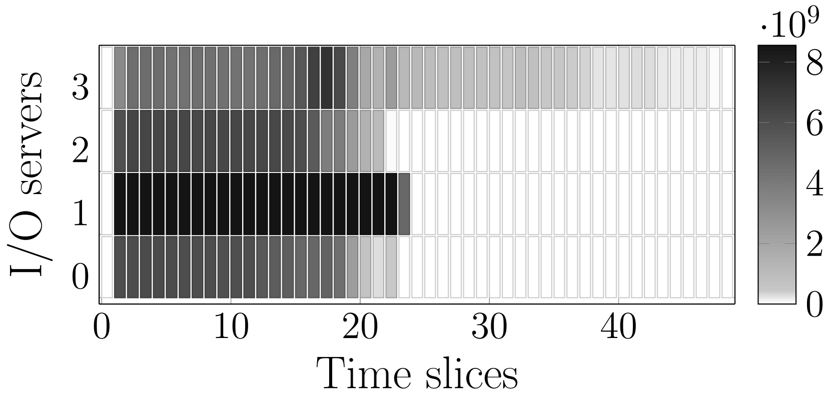


Figure 4. Write throughput of I/O servers. The performance of server 3 is degraded, leading to a longer execution time of I/O operations and hence the application

3.1. Environment

The results were obtained on the TSUBAME 2.5 supercomputer hosted at Tokyo Institute of Technology, Japan. The cluster comprises nodes in different configurations. The nodes used in our experiments make up the majority of the cluster and are equipped with two Intel Xeon X5670 (Westmere-EP, 2.93 GHz) 6-core processors, three NVIDIA Tesla K20X (GK110) GPUs, and 58 GiB DDR3 main memory. The cluster employs a two-rail fat-tree InfiniBand 4X QDR network, used both for message passing and file I/O traffic. The peak performance of the cluster is 2843 TFLOPS.

TSUBAME 2.5 offers GPFS and Lustre file systems for parallel I/O at different mount points, which are frequently updated. The configuration used in our experiments is as follows. GPFS on /data0 is hosted on four file servers (NSD servers), each connected to 14 RAID storage devices (NSDs), while Lustre on /work1 is hosted on eight file servers (OSSs), each of them connected to 13 RAID storage targets (OSTs). We used only these mount points in our experiments. On Lustre, metadata requests are handled by one MDS server with one additional standby server. The `qos_threshold_rr` parameter of Lustre has been set to 16%, meaning that storages are selected mostly in a round robin fashion. Additionally, TSUBAME 2.5 also provides 120 GB SSDs on compute nodes as scratch space. All file servers are equipped with two InfiniBand 4X QDR adapters, connecting them to one of the two rails of the fat-tree network. Table 1 provides a summary of the two file systems on the mount points we used.

Table 1. Specifications of the file systems on TSUBAME 2.5 used in our experiments

PFS	Mount point	Metadata server	File server	disks per server	Bandwidth
GPFS	/data0	N/A	4	14	20 GB/s
Lustre	/work1	1	8	13	50 GB/s

3.2. Experimental Setup

Except for the experiments comparing patterns at different process counts, a single instance of a mirco-benchmark or an application consisted of 256 processes, utilizing 64 compute nodes.

As the experiments were carried out on a production system, we took care of filtering out runs with more than 10% external noise. The filtering was done using the I/O server monitoring module of LWM^2. We also repeated each experiment five times and took the best-performing run (i.e., with the lowest degree of external interference).

We executed patterns with file sizes ranging from 1 MiB to 256 MiB on a logarithmic scale. For the open-write-close pattern and write-seek pattern, this meant that a file of the specified size was written repeatedly, while for the aggregate-write pattern this meant that each write operation had the specified buffer size.

3.3. Micro-benchmarks

In order to understand the interaction of different I/O access behaviors, we first paired up the three access patterns to form a collection of interference scenarios. We ran each of the three patterns against itself and against the other two, resulting in six experiments. For the purpose of interference quantification, however, we had to consider each micro-benchmark once as a signal and once as a probe, resulting in a total number of nine scenarios (i.e., $\{OWC, WS, AW\}^2$).

Table 2. Write bandwidth observed in an experimental run on Lustre when probe open-write-close is exposed to three different signal patterns at a chunk size of 1 MiB

			Signal	
		Open-write-close	Write-seek	Aggregate-write
Standalone	Bandwidth [GB/s]	28.4	31.6	42.2
Signal	Bandwidth [GB/s]	16.1	19.8	3.8
	Degradation [%]	43.31	35.76	9.95
Probe	Bandwidth [GB/s]	16.3	16.8	9.6
	Degradation [%]	42.61	40.85	66.2

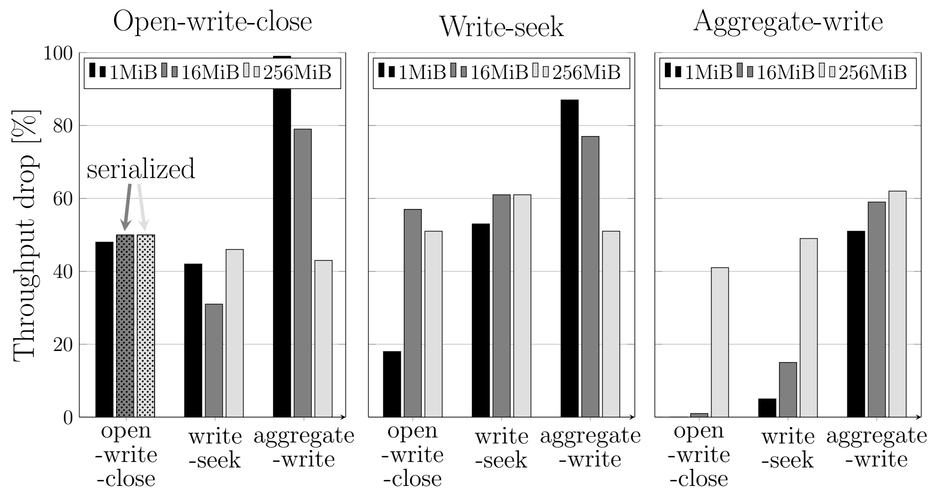
Table 2 shows the write throughput when an open-write-close probe is exposed to three different signal patterns. For both the probe and the signal patterns, we show the standalone and the interfered throughput. We also quantify severity of the interference effect in terms of the percentage degradation of the throughput T, defined as:

$$T = \frac{T_{standalone} - T_{interfered}}{T_{standalone}} \times 100.$$

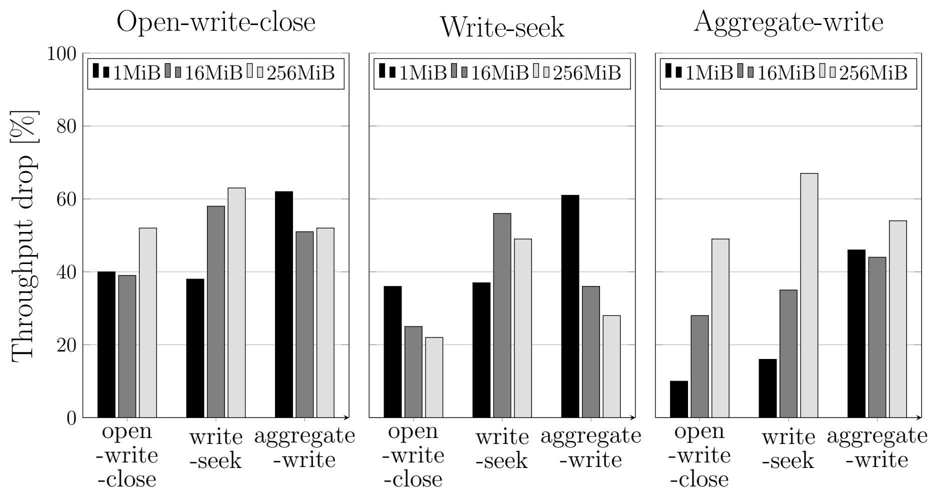
A high value of the degradation indicates severe interference inflicted by the signal pattern. As the focus of this paper is the severity of the interference, we restrict ourselves to relative throughput degradation figures in the remainder of the paper.

3.3.1. Access Patterns

We executed the complete set of combinations on both GPFS and Lustre for chunk sizes of 1 MiB, 16 MiB, and 256 MiB. Figure 5a shows a throughput drop observed with all pattern combinations, for chunk sizes of both 1 MiB, 16 MiB, and 256 MiB. With the smaller chunk sizes, we found aggregate-write to have a clearly higher interference potential than the other two



a) GPFS

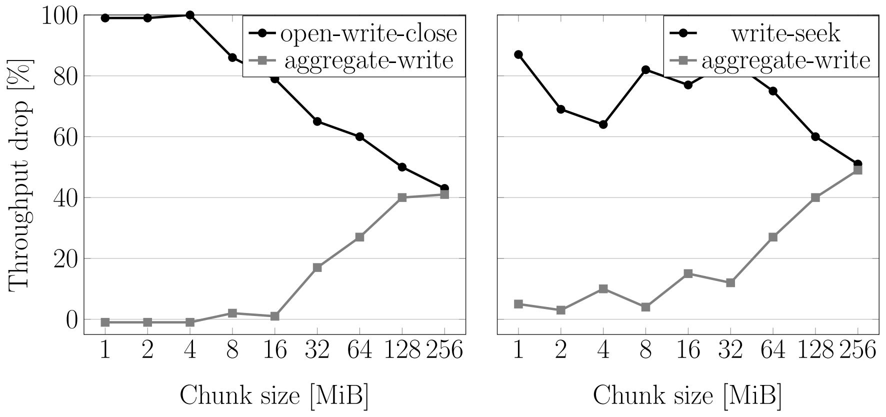


b) Lustre

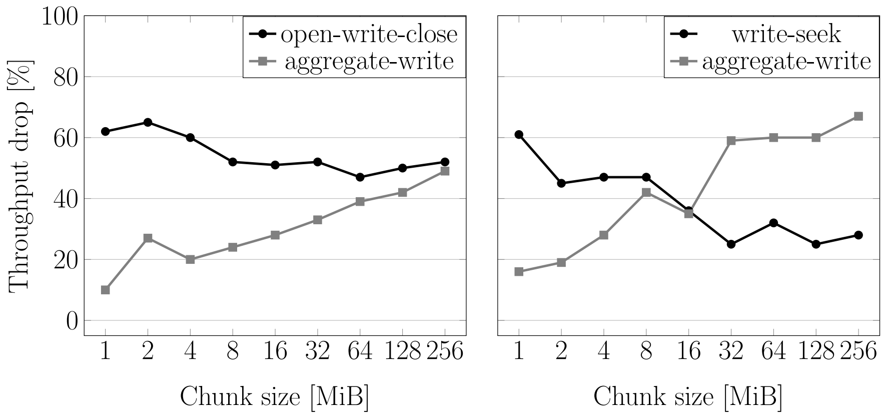
Figure 5. Throughput drop when the patterns are executed against each other. A higher bar means lesser throughput and higher passive interference. The top patterns indicate the probe, while the patterns below the x-axis indicate the signal

patterns. When open-write-close and write-seek are executed against each other, their throughput drops by about 45% to 55%. This can be explained by the equal sharing of I/O resources between them. However, concurrent execution of aggregate-write against the other two patterns reduces the latter's throughput by more than 80%, while the effect of the two other patterns on aggregate-write itself is much smaller. This indicates that aggregate-write dominates these two patterns at a chunk size of 1 MiB and 16 MiB, occupying most of the I/O resources. At a chunk size of 256 MiB, the I/O resources are distributed more evenly among the patterns. It can be seen that open-write-close is less affected by aggregate-write compared to the 1 MiB case, while aggregate-write is more affected by the other two patterns. Open-write-close, at chunk sizes 16 MiB and 256 MiB, when executed against itself, becomes serialized, that is, one pattern of the pair executes first, almost completely degrading the second pattern during first's execution.

We repeated the same set of experiments on Lustre. The results from the nine pair-wise combinations of patterns for 1 MiB, 16 MiB, and 256 MiB are shown in Fig. 5b. The general trend of the interference potential for the three chunk sizes is the same as on GPFS but with different intensities. At a chunk size of 1 MiB, aggregate-write generates most of the interference,



a) GPFS



b) Lustre

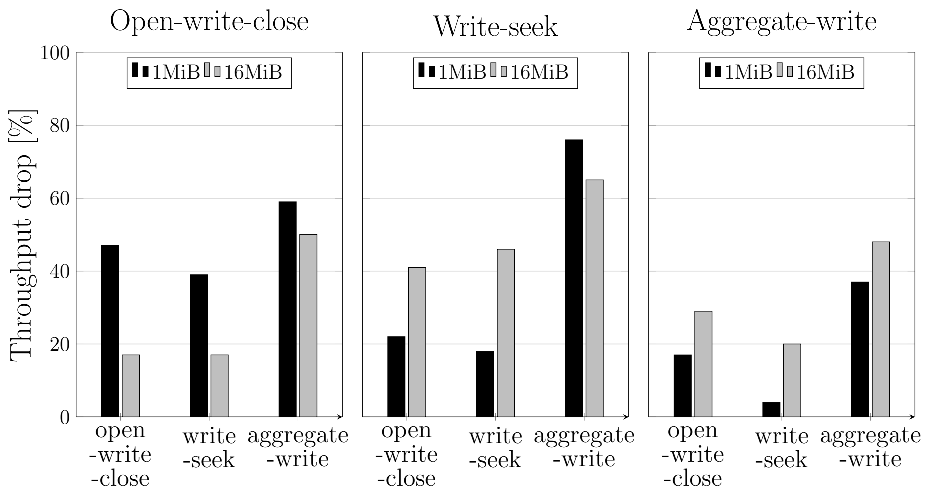
Figure 6. Effect of chunk size on throughput degradation

again while itself being the least affected one. However, the disparity is not as strong as on GPFS. As the chunk size is increased to 16 MiB and 256 MiB, respectively, the interference potential of aggregate-write decreases, whereas that of open-write-close and write-seek increases. At a chunk size of 256 MiB, write-seek causes most of the throughput reduction, more than 60% for the other two patterns.

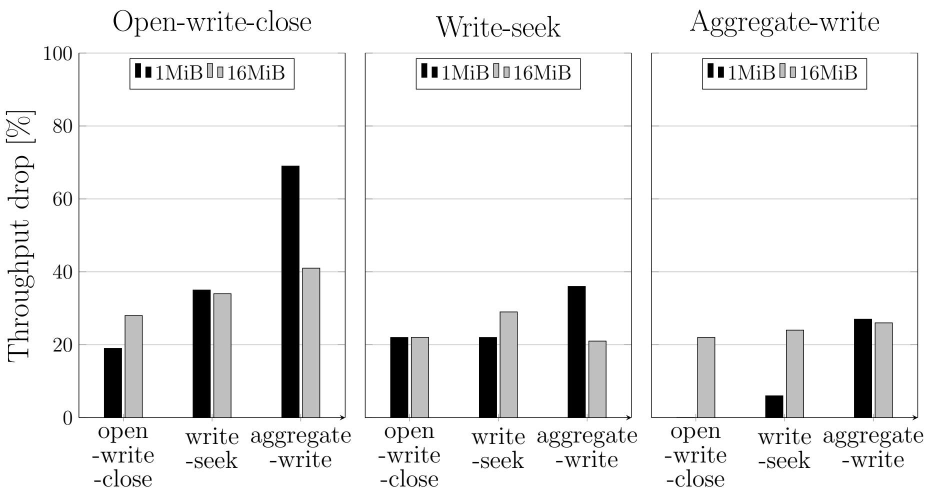
3.3.2. Chunk Size

As chunk size seems to be a crucial parameter for the interference potential of the above mentioned patterns, we investigated this more closely by running open-write-close and write-seek against aggregate-write for chunk sizes ranging from 1 MiB to 256 MiB on a logarithmic scale. The results for GPFS are shown in Fig. 6a. Open-write-close seems to share I/O resources with aggregate-write more evenly as the chunk size increases, with 256 MiB being the break-even point. Write-seek shows a similar trend but with the slope shifted to the right. The convergence here begins when a chunk size of 32 MiB is reached. Beyond this point, the progression is similar to open-write-close, as I/O resources start to be shared more evenly. At the last data point of 256 MiB, the two patterns break even.

We also studied the sensitivity of interference to chunk sizes on Lustre. The results are summarized in Fig. 6b. The trend of open-write-close on GPFS, where, at small chunk sizes, aggregate-write dominates over open-write-close, reappears on Lustre. As the chunk size in-



a) GPFS



b) Lustre

Figure 7. Throughput drop when the patterns are executed against each other. A higher bar means less throughput and a higher passive interference. The pattern above each chart represents the probe, whereas the patterns below the x-axis represent the signal. The signal pattern is executed in a periodic fashion

creases, open-write-close starts to perform better. The trend culminates at 256 MiB, where open-write-close and aggregate-write experience the same amount of throughput drop. In the case of write-seek, aggregate-write dominates at small chunk sizes. However, as the chunk size is increased, the trend is quickly reversed. Both patterns suffer the same amount of throughput degradation at 16 MiB, beyond which write-seek starts to dominate aggregate-write.

3.3.3. High Frequency vs. Low Frequency

As the sensitivity to chunk size shows, the trend of interference among the patterns depends on their specific characteristics. To evaluate this further, we consider the file access frequency of a pattern. However, covering the whole breadth of possible write access frequencies is prohibitively expensive. Instead, we expose the unaltered probe to a periodic signal, in which write activity phases alternate with computational busy-wait phases, mimicking bursty I/O. The period length of the signal was set to 24 seconds so that multiple consecutive time slices of LWM^2 fall in one

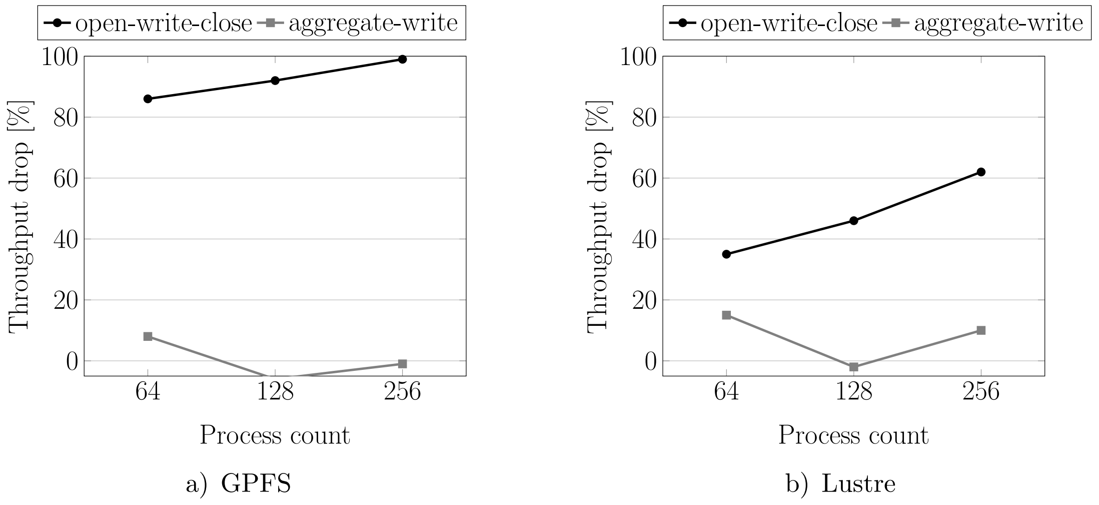


Figure 8. Effect of process count on the passive degradation produced by patterns on GPFS and Lustre

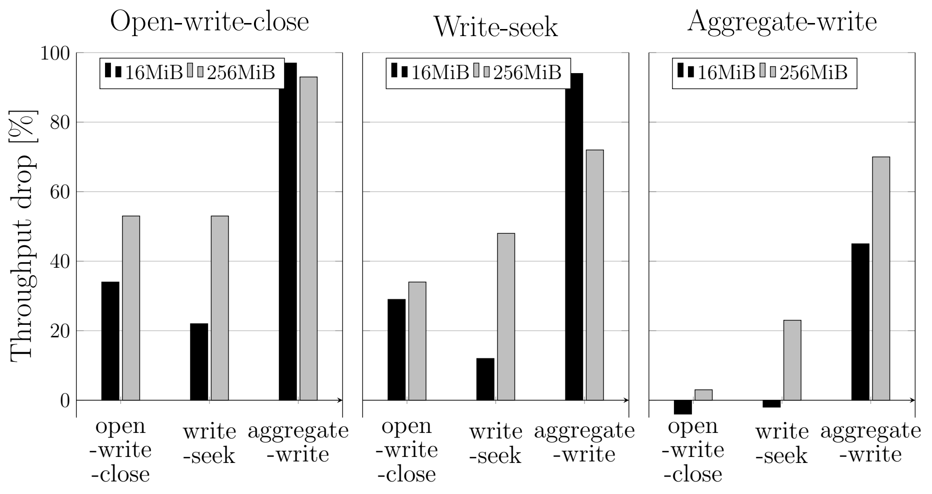
period. Note that the write activity phases are repeated multiple times in a run. The results of the experiments are shown in Fig. 7.

Overall, the interference trend, both for GPFS and Lustre, is similar to what was observed for the sustained activity patterns, but with a lower intensity. Aggregate-write still dominates over open-write-close and write-seek at a chunk size of 1 MiB. Similarly, at the larger 16 MiB chunk size, aggregate-write causes a lesser degradation while finding itself being victimized to a higher degree. On GPFS, at a chunk size of 16 MiB, open-write-close suffers less throughput degradation against itself and against write-seek. One of the reasons it can be like this is at lower frequencies and at larger write chunk size, is that the metadata operations cease to be the I/O bottleneck, while, additionally, the low frequency prevents the write bandwidth of the system from being saturated. As a result, the performance of open-write-close degrades to a lesser degree.

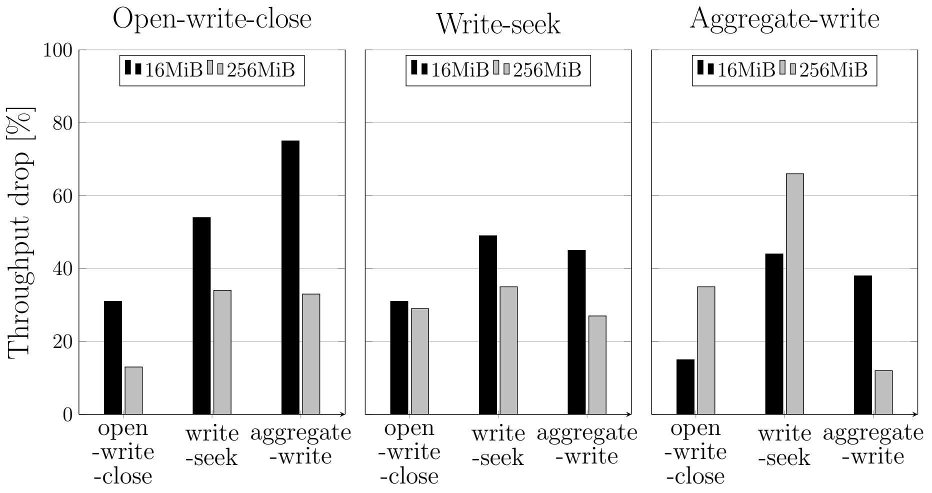
3.3.4. Process Count

It has been previously observed that an application with higher process count dominantly occupies the I/O resources of a system when run against an application with lower process count [29]. However, does this relationship hold true if the two applications have different file access patterns? We investigated this by running open-write-close and aggregate-write against each other with a chunk size of 1 MiB. Because write-seek is similarly dominated by aggregate-write at this chunk size, we have concentrated our study on open-write-close. For each run, we executed the open-write-close pattern with 256 processes and the aggregate-write pattern with 64, 128 and 256 processes. The results of the experiments are shown in Fig. 8.

The blue line shows the throughput degradation of open-write-close while the red line shows the throughput degradation of aggregate-write. For GPFS, we see in Fig. 8a that open-write-close is degraded severely, even when aggregate-write occupies only one fourth of the space. As we increase the process count of aggregate-write, open-write-close degrades even more severely. Figure 8b shows the trend for Lustre, which is similar to that of GPFS, but with a lesser degradation. Again, we see that open-write-close suffers higher degrees of degradation when run against aggregate-write, even when aggregate-write is one fourth of the size. The throughput



a) GPFS



b) Lustre

Figure 9. Throughput drop in MPI shared-file mode when the patterns are executed against each other. A higher bar means less throughput and higher passive interference. The pattern above each chart represents the probe, whereas the patterns below the x-axis represent the signal

of open-write-close decreases even further as the process count of aggregate-write increases. From these experiments we can conclude that, when it comes to sharing I/O resources between applications, the write-access pattern can play a bigger role than the application process count.

3.3.5. Shared File

Not to ignore this increasingly common mode, we also performed a set of experiments on shared files. The file was shared in such a way that each process occupied a contiguous portion of the file. For open-write-close and write-seek, the size of the contiguous portion exactly matched the chunk size of the benchmark. For the aggregate-write pattern, the contiguous portion matched the size of the total data written by a process.

Figure 9a shows the interference potentials on GPFS. At a chunk size of 16 MiB, aggregate-write dominates the other two patterns significantly, while at 256 MiB, even though being still dominant, it generates comparatively less interference for other patterns. These observations are similar to the results with one file per process. However, we observed some cases in the pairwise execution of the patterns, in which one instance would completely dominate over the other instance, effectively serializing the I/O traffic between the pairs. This near-serialization of the I/O traffic was observed when running patterns against themselves as well as against different patterns. Figure 9b presents the results on Lustre, where we observed that aggregate-write dominates open-write-close at a chunk size of 16 MiB, while becoming slightly dominated by write-seek itself. At a chunk sizeof 256 MiB, open-write-close and write-seek are evenly interfered in all the runs but dominate aggregate-write. The behavior of aggregate-write is again consistent with the one-file-per-process case. Open-write-close, however, is less prone to interference at larger chunk sizes. Based on these observations, and considering that writing shared files is a topic of research in its own right with its own characteristic set of access patterns, we believe that a full coverage of shared files would justify a separate study.

3.3.6. Discussion

From the above results, it is clear that different I/O access patterns show different interference potential. The chunk size is also an important factor in determining which pattern is dominant. At smaller chunk sizes, aggregate-write prevails over open-write-close and write-seek, causing a notable degradation of throughput for the latter two while showing little impact on the former. However, as the chunk size increases, the balance is shifted in favor of open-write-close and write-seek. At a certain point, open-write-close and write-seek suffer as much as aggregate write, beyond which the trend may even become reversed. On GPFS, open-write-close and write-seek show similar degradation trends, while on Lustre, open-write-close has comparatively less interference potential. The precise reason for our observations is unclear, but it seems that both metadata operations including open, close, and seek on the one hand and the number of different file blocks an application writes make it sensitive for interferention. At least, this would explain the trend reversal shown in Fig. 6b. As the chunk size, increases together with it the number of different blocks written by aggregate-write, the density of metadata operations shrinks.

3.4. Applications

After establishing an interference relationship among the access patterns through micro-benchmarks, we investigated the same effects using realistic applications. First, we verified the interference trend for micro-benchmarks against applications, and later confirmed it for application vs application. In our previous study [6], we considered two typical I/O-intensive HPC applications, OpenFOAM and MadBench2. Here, we extend the work by also evaluating our approach with HACCIO, a code-writing large checkpoints, against the micro-benchmarks and the other two applications. All three together of them provide one realistic use for each of the three access patterns, as summarized in Tab. 3.

Table 3. Applications and the access pattern they represent including the chunk size

Application	Access pattern	Chunk size
OpenFOAM	open-write-close	a few kilobytes
MADBench2	write-seek	74 MB
HACCIO	aggregate-write	386 MB

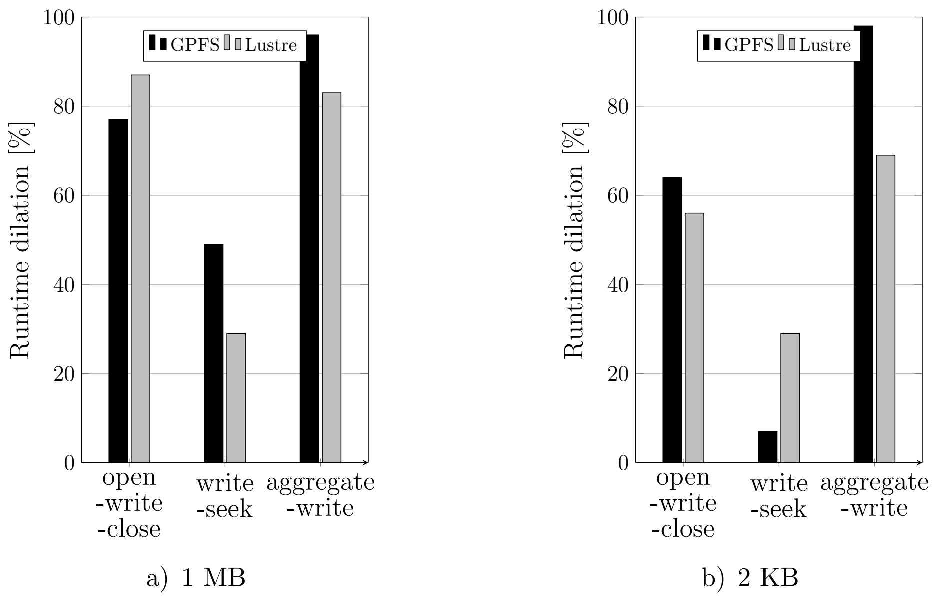


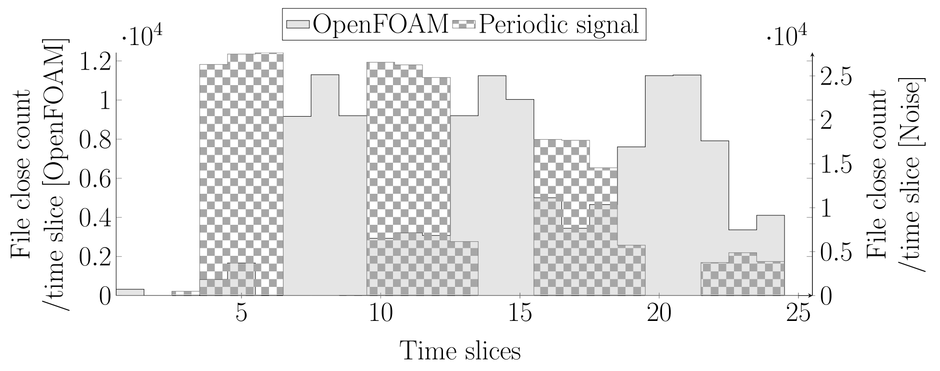
Figure 10. Throughput degradation of OpenFOAM when run against the patterns at different chunk sizes

3.4.1. OpenFOAM

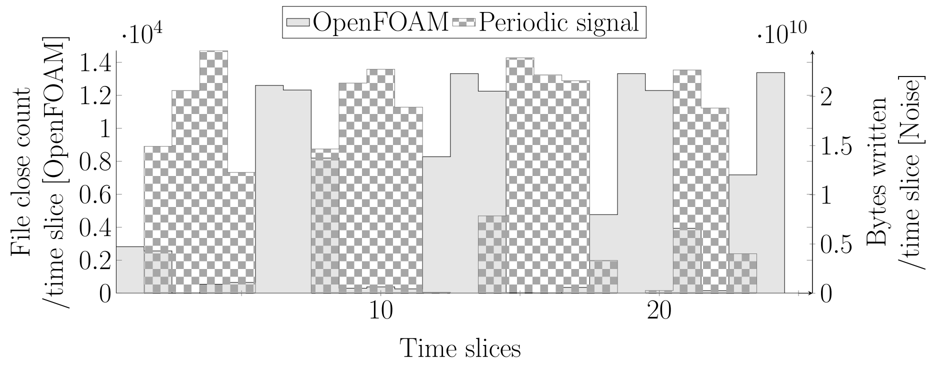
OpenFOAM [17], which stands for Open source Field Operation And Manipulation, is a free, open source computational-fluid-dynamics (CFD) software package developed by OpenCFD Ltd of ESI Group and distributed by the OpenFOAM Foundation. It was one of the first scientific applications to leverage C++ for a modular design. The package provides parallel implementations of a rich set of libraries, from mathematical equation solvers to general physical models. OpenFOAM uses standard C++ I/O for checkpointing at regular intervals. At each checkpoint, new files of around a few kilobytes are created and written by every process, making its I/O behavior similar to the open-write-close pattern. As LWM^2's C++ I/O profiling is still in progress, we were only able to capture file-close counts for our runs. In our experiments, OpenFOAM closed more than 14000 files per time slice. As this count is significantly larger than the process count of the application, it indicates that most of those files were written to and closed in the same time slice, making the file-close count an indicator of I/O throughput. Similarly, we used the dilation of execution time, which occurs as a consequence of I/O performance drop, to gauge the interference potential.

In our experiment, we ran the cavity example from the official tutorial of OpenFOAM version 2.3.0 using 256 processes. Cavity involves processing of an isothermal, incompressible flow in a two-dimensional square domain. Specifically, we used the icoFoam solver, in which the flow is assumed to be laminar. We executed the cavity example in parallel with each of the three pattern micro-benchmarks. We set the chunk size of the patterns to 1 MiB, the smallest chunk size we used in our pure micro-benchmark experiments. We executed the runs on both GPFS and Lustre, and adjusted the runtime of the patterns to fully overlap with OpenFOAM's execution.

OpenFOAM experienced degraded I/O performance when executed concurrently with all the three patterns. The throughput drop caused by each of the patterns is shown in Fig. 10a. As OpenFOAM's I/O pattern is similar to open-write-close with a small chunk size, the large interference potential of aggregate-write at such a small chunk size is immediately visible, leading



a) OpenFOAM vs. open-write-close



b) OpenFOAM vs. aggregate-write

Figure 11. Time-slice view of OpenFOAM when executed concurrently with two patterns on GPFS

to more than 80% drop for Lustre and 90% for GPFS. Unlike the micro-benchmark, OpenFoam also suffered about 80% throughput drop against open-write-close. One reason for this behavior might be the unequal chunk size of the patterns and OpenFOAM. To verify this, we executed the patterns at 2 KiB chunk size. The results are shown in Fig. 10b. Aggregate-write still dominates over OpenFOAM, with a throughput drop of more than 90% for GPFS. Lustre, on the other hand, shows a slightly reduced drop of 70% in throughput. Open-write-close now degrades OpenFOAM's throughput by around 60%, similar to what the micro-benchmark allowed us to see. The interference of write-seek on Lustre remains at 30% for both chunk sizes. However, it declines from around 50% for 1 MiB to 10% for 2 KiB on GPFS. Overall, the interference trend is similar to that of our purely micro-benchmark-based observations.

To further understand the I/O interference dynamics during concurrent execution, we executed OpenFOAM against periodic modes of open-write-close and write-seek. In this mode, the micro-benchmark's I/O access phases alternate with silence. This periodic mode highlights the effects of interference during the I/O access phases. Figure 11a and Figure 11b show the *time slice view* when OpenFOAM is concurrently executed with open-write-close and aggregate-write, respectively. Against open-write-close, OpenFOAM's performance degrades by 60%–70% during active phases of the pattern in comparison to the silent phases. On the other hand, OpenFOAM against aggregate-write degrades by up to 95% when the pattern performs I/O accesses. This is clearly visible, as the file-close rate of OpenFOAM exhibits intermittent behavior under interference. Comparing OpenFOAM to our open-write-close micro-benchmark, we see that it suffers in a similar way when exposed to aggregate-write, that is, its performance degrades significantly.

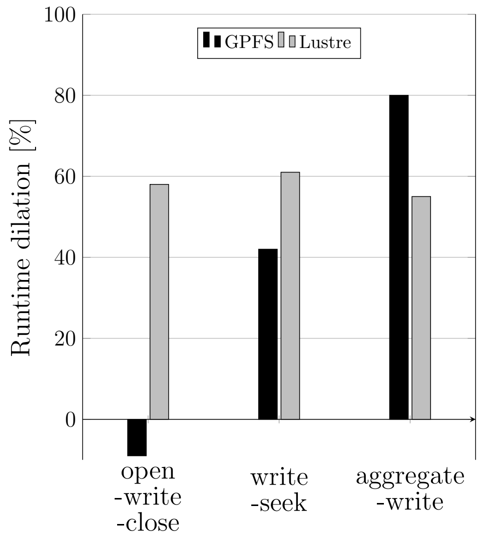


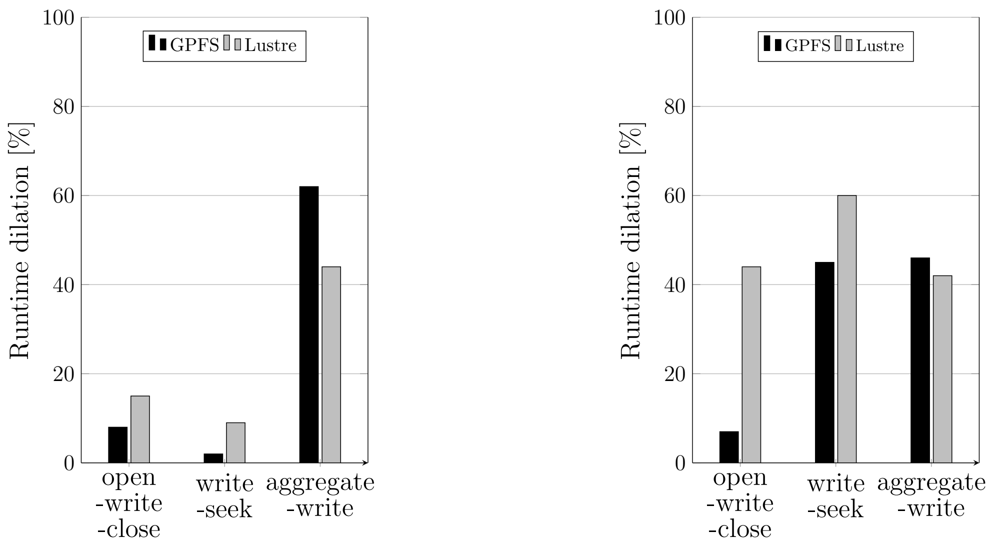
Figure 12. Throughput degradation of MadBench2 when run against different patterns

3.4.2. MadBench2

MADbench2 is derived from MADCAP cosmic microwave background radiation analysis software. MADbench2 performs dense-linear-algebra calculation using ScaLAPACK [13]. It has very large memory demands and its required matrices generally do not fit in the memory. As a result, the calculated matrices get recorded to a disk and re-read when required. This means that MadBench2 performs complex I/O operations in four phases. For our experiments, as the scope of our study is write-write contentions, we concentrate on the first phase, which has only writes and seeks. The other phases are either reads or a mixture of reads and writes. We henceforth use MadBench2 to refer to the build with the first phase only.

For the experiments, we setup MadBench2 to use POSIX I/O in the one-file-per-process mode. To maximize performance, we used the configuration recommended by Borill et al. [34], which is: WMOD=1, NPIX=50,000, NBIN=36, NGANGS=1, SBBLOCKSIZE=1, FBBLOCKSIZE=128. Furthermore, as our focus is on file-access patterns, MadBench2 is configured to run in I/O mode. In I/O mode, MadBench2 acts as a pure I/O benchmark, replacing computation with busy-wait cycles. With this configuration, and using 256 processes, MadBench2 writes 670 GB of data, with each process, performing seeks with an offset of about 74 MB during execution. This makes the I/O behavior similar to the write-seek pattern with a chunk size of 74 MB. For this reason, we executed MadBench2 against the three patterns at a chunk size of 64 MB. The results are shown in Fig. 12.

The throughput degradation on both file systems is quite different for MadBench2. On GPFS, aggregate-write generates the most interference, reducing the throughput by about 80%. Similarly, write-seek degrades the throughput of MadBench2 by about 40%. However, in the case of open-write-close, MadBench2's runtime improves. For a chunk size of 64 MiB, the higher interference aggregate-write generates is consistent with our micro-benchmarks results. On Lustre, all the patterns generate similar interference levels, with aggregate-write degrading the throughput of MadBench2 slightly less compared to others. The reason is that, for write-seek, the interference trend already reverses at a chunk size of 64 MiB, as shown in Fig. 6b. Overall, the passive interference behavior of MadBench2 resembles that of our write-seek micro-benchmark.



a) 1 MB b) 256 MB

Figure 13. Throughput degradation of HACCIO when run against different patterns

3.4.3. HACCIO

HACCIO is an I/O benchmark code derived from a cosmology software framework called HACC (Hardware Accelerated Cosmology Code). HACC simulates the formation of collisionless fluids under the influence of gravity using N-body techniques. HACC has very high I/O demands, where a small simulation can write terabytes of data [7].

HACCIO writes large checkpoint files during its execution. It creates one file per process, and incrementally writes data to it. During checkpointing, the files are written, read back, and verified. As our work concentrates on write-write contention, we removed the read-back and verification part in our experiments. HACCIO can use different I/O modes during execution, including POSIX I/O, MPI with one-file-per-process or MPI with one or more shared-file.

In our experiments, we ran HACCIO with 256 processes and with POSIX I/O. During execution, each process wrote 3.6 GiB of data to its file, in chunks of 381 MiB. The I/O behavior can be equated to the aggregate-write pattern with a chunk size of 381 MiB. We ran HACCIO against the three patterns with chunk sizes of 1 MiB and 256 MiB, respectively. The results are shown in Fig. 13.

With 1 MiB, on both GPFS and Lustre, open-write-close and write-seek degrade HACCIO's performance to a smaller degree than aggregate-write. This trend is consistent with our micro-benchmark results. In the case of aggregate-write, the degradation that HACCIO suffers on GPFS is slightly higher than the one on Lustre (60% vs. 40%). In our micro-benchmark experiments for 1 MiB and 16 MiB, we also saw aggregate-write suffering a degradation of around 40% on Lustre and one between 50% and 60% on GPFS. With 256 MiB on Lustre, the degradation caused by write-seek grows to about 60%, while open-write-close and aggregate-write cause around 50% degradation. This is again similar to what has been observed with micro-benchmarks. Write-seek dominates aggregate-write at large chunk sizes. On GPFS, open-write-close degrades HACCIO by less than 10%. On the other hand, write-seek and aggregate-write cause about 50% degradation of the HACCIO write throughput. Overall, the trend is similar to our micro-benchmark-based observations.

3.4.4. Application vs. Application

With our knowledge of how isolated access patterns interfere with realistic applications, we also investigated the interference between realistic applications, as it can occur in a live production system. For this purpose, we ran OpenFOAM, MADBench2, and HACCIO first against themselves and later against each other, always using 256 processes per application. The results are shown in Fig. 14. In the figure, the x-axis shows the probe applications whose runtime dilation is reported. For each probe application, we show a separate bar for each signal application that is causing a degraded performance. In these figures, each application represents an access pattern; however, each one of them has a different chunk size and access frequency. Therefore, the interpretation of our results requires consideration of the pattern type, chunk size, and access frequency.

On GPFS, HACCIO generates the biggest interference of all, with OpenFOAM being degraded by more than 90% and MadBench2 by more than 60%. The values are similar to open-write-close against aggregate-write at a chunk size of 1 MiB and write-seek against aggregate-write at 256 MiB. MadBench2 degrades OpenFOAM by more than 80% and HACCIO by more than 10%. Here, MadBench2's behavior diverges from write-seek, with high degradation for OpenFOAM and low degradation for HACCIO. A possible explanation for OpenFOAM against MadBench2 can be the large chunk-size difference, while for HACCIO it can be low access frequency, as was observed for periodic probe signals in Fig. 7a. OpenFOAM against the other two applications generates a comparatively small throughput degradation. This is similar to our observation of open-write-close at small chunk sizes.

On Lustre, HACCIO degrades OpenFOAM by about 60%, while being degraded itself by less than 10%, similar to what was observed with micro-benchmarks. HACCIO degrades MadBench2 by about 55%, while being degraded itself by about 40%. This is again similar to micro-benchmark results, where for chunk sizes greater than 16 MiB, write-seek dominates over aggregate-write. Looking at MadBench2 against OpenFOAM, we see that OpenFOAM's runtime is dilated by about 70%. This is because of the large chunk-size difference between OpenFOAM and MadBench2.

Considering the different access patterns, write chunk sizes, and access frequencies, the overall results are in line with our observations of synthetic micro-benchmarks.

4. Related Work

Several earlier studies identified typical I/O access patterns of HPC applications. Miller et al. found I/O to be bursty and cyclic [23]. They also distinguished three access patterns, namely required I/O, checkpointing, and data staging as the most common I/O types. These patterns roughly correspond to our aggregate-write, open-write-close, and write-seek patterns, respectively. However, they were studied to optimize I/O from a single-application perspective, while we look at their interference potential when executed concurrently.

Byna et al. classified file access patterns to generate I/O-access signatures of applications [12]. These signatures were then used to improve data prefetching. Shan et al. created a parameterized I/O benchmark called IOR that can mimic the file access pattern of realistic applications [14]. Lofstead et al. found six common read patterns in the analysis part of simulation software [37]. The read patterns were used to compare end-to-end performance of logically contiguous and log-based files. Congiu et al. manually analyzed the I/O behavior of

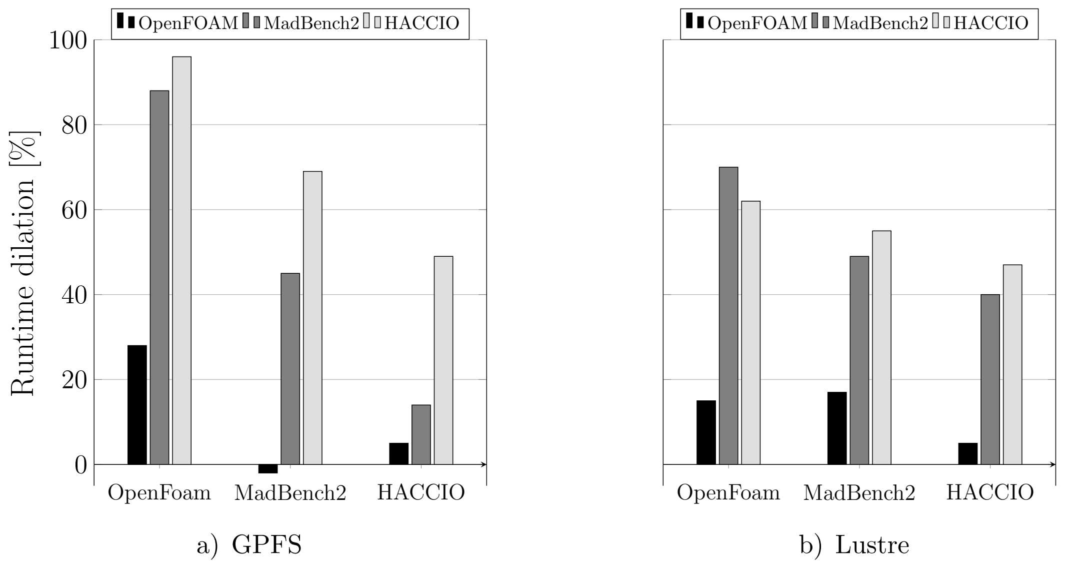


Figure 14. Throughput degradation when the applications are run against each other

applications to identify their patterns [20]. A framework, transparent to the application, then translated the knowledge of these patterns into hints to the parallel file system. Lu et al. analyzed patterns in collective I/O and found that the access pattern of a process can be lost after aggregation, negatively impacting cache performance [1]. To mitigate this effect, they proposed a cache-management policy aware of collective I/O. In our work, we evaluate the interference potential of I/O access patterns in concurrent execution.

Similarly, as part of the SIO initiative, Smirni et al. classified I/O accesses according to their spatial and temporal patterns [18, 25]. Nieuwejaar et al. classified file accesses with respect to access size, file size, access frequency, sequentiality, etc. in the CHARISMA project [39]. These studies are orthogonal to our work and part of the broader field of file-access characterization. More recent work on the topic includes characterizing read access patterns of applications with the goal of optimizing reads for subsequent data analysis and visualization [37]. Liu et al. analyzed server-side logs to identify I/O intensive applications and characterize their workloads, providing recommendation to an I/O-aware scheduler [4]. Our work studies the effects of write patterns on the I/O performance of other co-scheduled applications.

I/O performance has been the subject of several studies, looking at the performance from a single application perspective [34], from the file-system perspective [26], and from the overall system perspective [2, 11]. Further studies considered I/O interference between different jobs, identifying variability [36], uncovering performance problems with statistical techniques [32], and mitigating I/O interference through application coordination and scheduling [29]. In this paper, we analyze how file writes of concurrently running jobs interfere and determine factors that influence the magnitude of interference. While the application process count is already known as one of the factors [29], we consider process count in the context of access patterns and examine the influence of further parameters such as write-chunk size and access frequency on write performance.

SIOX records I/O accesses at each level of the I/O stack, identifies access patterns, and characterize the I/O subsystem [27, 38] with the objective of pinpointing I/O bottlenecks. Our work contributes insights into write performance variation as a result of access patterns and request sizes.

Yildiz et al. studied the root cause of inter-application I/O interference in HPC storage systems by comparing the impact of different factors [19]. They found that bad flow of control in the I/O path caused interference in most cases. Whereas they looked at I/O interference from the storage perspective, this paper takes an application-centric view.

Inter-application interference in general has also been subject of several studies. Skinner et al. identified it as one of the five sources of performance variability [31]. Shah et al. established a framework for correlating application performance across job boundaries and found I/O to be highly susceptible to the overall system load [9]. Bhatele et al. observed communication performance to be strongly influenced by co-scheduled applications on Hopper, a Cray XE system [35]. Finally, Shah et al. developed a framework to estimate the impact of inter-application interference on the execution time of bulk-synchronous MPI applications [10].

Several tools have been used to profile and monitor I/O performance of applications. Carns et al. used Darshan to characterize I/O of applications at the system level [15, 16]. Uselton et al. extended IPM for their statistical study of I/O performance variation [32]. We used LWM^2 for our study because of its ability to generate synchronized, segmented profiles that allow the performance of co-scheduled applications to be precisely correlated [9].

Conclusion

In this study, we analyzed inter-application interference effects caused by the interaction between various I/O access patterns, classified by their behavior, write chunk size, access frequency, process count, and sharing mode. Specifically, we found that at small chunk sizes data-intensive applications may significantly slow down checkpointing-intensive applications, even at smaller process counts, but not vice versa. In one case, the runtime of a checkpointing-intensive application was dilated by a factor of five. But the direction of the interference is continuously reversed as the chunk size is increased.

Given the shared nature of the majority of parallel file systems, preventing I/O interference altogether is challenging. As a general strategy to reduce it, one should try to separate I/O traffic with high interference potential either in space or in time. However, in order to make such a separation successful, it is important to decide what traffic should be separated. Leveraging techniques demonstrated now with LWM^2, file systems could be extended in the future to recognize aggressive or sensitive patterns automatically, and dynamically separate them either in space or in time. For example, traffic to a specific set of files could be (re-)routed to a specific group of file servers or buffered locally to be written back at a later point in time.

In order to support the future interference-aware file-system designs, we plan to further extend LWM^2 to recognize application I/O access patterns automatically and suggest some appropriate I/O resource scheduling policies. To this end, we want to take more complicated patterns, chunk sizes, and I/O frequencies into account with the objective of building a reliable I/O performance interference model based upon quantifiable application I/O characteristics. The interference model would also pay attention to higher-level file formats such as NetCDF and HDF5. Finally, with LWM^2's global time-slice view and the ability to detect interference through correlation, we also see machine learning techniques as a promising research direction for the prediction of interference and ultimately for its avoidance.

Acknowledgements

This research was supported by JST, CREST (Research Area: Advanced Core Technologies for Big Data Integration) Grant No. JPMJCR1303 and the G8 Research Councils Initiative on Multilateral Research, Interdisciplinary Program on Application Software towards Exascale Computing for Global Scale Issues. Additional support was provided by the German Research Foundation (DFG) through the program Performance Engineering for Scientific Software and the US Department of Energy under Grant No. DE-SC0015524. Finally, we would like to acknowledge the support of Global Scientific Information and Computing Center at Tokyo Institute of Technology for giving us access to their system.

References

1. Lu, Y., Chen, Y., Latham, R., Zhuang, Y.: Revealing applications' access pattern in collective I/O for cache management. In: Proceedings of the 28th ACM International Conference on Supercomputing, ICS, Munich, Germany, June 10-13, 2014. pp. 181–190. ACM (2014), DOI: 10.1145/2597652.2597686
2. Yu, W., Vetter, J., Oral, H.: Performance characterization and optimization of parallel I/O on the Cray XT. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, IPDPS, Miami, FL, USA, April 14-18, 2008. pp. 1–11. IEEE Computer Society (2008), DOI: 10.1109/IPDPS.2008.4536277
3. IBM: An Introduction to GPFS Version 3.5. `http://www-03.ibm.com/systems/resources/introduction-to-gpfs-3-5.pdf` (2014), accessed: 2014-08-11
4. Liu, Y., Gunasekaran, R., Ma, X., Vazhkudai, S.S.: Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC16, Salt Lake City, UT, USA, November 13-18, 2016. pp. 819–829. IEEE Computer Society (2016), DOI: 10.1109/SC.2016.69
5. Xie, B., Chase, J., Dillow, D., Drokin, O., Klasky, S., Oral, S., Podhorszki, N.: Characterizing output bottlenecks in a supercomputer. In: Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis, SC'12, Salt Lake City, UT, USA, November 10-16, 2012. pp. 8:1–8:11. IEEE Computer Society (2012), DOI: 10.1109/SC.2012.28
6. Kuo, C.S., Shah, A., Nomura, A., Matsouka, S., Wolf, F.: How file access patterns influence interference among cluster applications. In: Proceedings of the IEEE International Conference on Cluster Computing, CLUSTER, Madrid, Spain, September 22-26, 2014. pp. 1–8. IEEE Computer Society (2014), DOI: 10.1109/CLUSTER.2014.6968743
7. Hal Finkel: Cosmic Structure Probes of the Dark Universe(Porting and Tuning HACC on Mira). `https://www.alcf.anl.gov/files/darkuniverseesptechreportwrapped.pdf` (2014), accessed 2014-08-11

8. The National Institute for Computational Sciences: I/O and Lustre Usage. `https://www.nics.tennessee.edu/computing-resources/file-systems/io-lustre-tips` (2014), accessed: 2014-08-11

9. Shah, A., Wolf, F., Zhumatiy, S., Voevodin, V.: Capturing inter-application interference on clusters. In: Proceedings of the IEEE International Conference on Cluster Computing, CLUSTER, Indianapolis, IN, USA, September 23-27, 2013. pp. 1–5. IEEE Computer Society (2013), DOI: 10.1109/CLUSTER.2013.6702665

10. Shah, A., Müller, M.S., Wolf, F.: Estimating the impact of external interference on application performance. In: Proceedings of the Euro-Par 2018: Parallel Processing, Turin, Italy, August 27-31, 2018. Lecture Notes in Computer Science, vol. 11014, pp. 46–58. Springer, Cham (2018), DOI: 10.1007/978-3-319-96983-1_4

11. Lang, S., Carns, P., Latham, R., Ross, R., Harms, K., Allcock, W.: I/O performance challenges at leadership scale. In: Proceedings of the ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis, SC'09, New York, NY, USA, November 14-20, 2009. pp. 40:1–40:12. IEEE Computer Society (2009), DOI: 10.1145/1654059.1654100

12. Byna, S., Chen, Y., Sun, X.H., Thakur, R., Gropp, W.: Parallel I/O prefetching using MPI file caching and I/O signatures. In: Proceedings of the ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis, SC'08, Piscataway, NJ, USA, November 15-21, 2008. pp. 44:1–44:12. IEEE Computer Society (2008), DOI: 10.1109/SC.2008.5213604

13. Choi, J., Dongarra, J.J., Pozo, R., Walker, D.W.: ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers. In: Proceedings of the IEEE Fourth Symposium on the Frontiers of Massively Parallel Computation, McLean, VA, USA, October 19-21, 1992. pp. 120–127. IEEE Computer Society (1992), DOI: 10.1109/FMPC.1992.234898

14. Shan, H., Antypas, K., Shalf, J.: Characterizing and predicting the I/O performance of hpc applications using a parameterized synthetic benchmark. In: Proceedings of the ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis, SC'08, Austin, TX, USA, November 15-21, 2008. pp. 42:1–42:12. IEEE Computer Society (2008), DOI: 10.1109/SC.2008.5222721

15. Carns, P., Harms, K., Allcock, W., Bacon, C., Lang, S., Latham, R., Ross, R.: Understanding and improving computational science storage access through continuous characterization. In: Proceedings of the IEEE 27th Symposium on Mass Storage Systems and Technologies, MSST, Denver, CO, USA, May 23-27, 2011. vol. 1, pp. 1–14. IEEE Computer Society (2011), DOI: 10.1109/MSST.2011.5937212

16. Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., Riley, K.: 24/7 characterization of petascale I/O workloads. In: Proceedings of the IEEE International Conference on Cluster Computing and Workshops, CLUSTER, New Orleans, LA, USA, August 31-September 04, 2009. pp. 1–10. IEEE Computer Society (2009), DOI: 10.1109/CLUSTR.2009.5289150

17. Jasak, H., Jemcov, A., Tukovic, Z.: OpenFOAM: a C++ library for complex physics simulations. In: Proceedings of the International workshop on coupled methods in numerical dynamics, IUC, Dubrovnik, Croatia, September 19-21, 2007. pp. 1–20 (2007)

18. Smirni, E., Aydt, R., Chien, A., Reed, D.: I/O requirements of scientific applications: an evolutionary view. In: Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing (HPDC'96), Syracuse, NY, USA, August 06-09, 1996. pp. 49–59. IEEE Computer Society (1996), DOI: 10.1109/HPDC.1996.546173

19. Yildiz, O., Dorier, M., Ibrahim, S., Ross, R., Antoniu, G.: On the root causes of cross-application I/O interference in HPC storage systems. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium, IPDPS, Chicago, IL, USA, May 23-27, 2016. pp. 750–759. IEEE Computer Society (2016), DOI: 10.1109/IPDPS.2016.50

20. Congiu, G., Grawinkel, M., Padua, F., Morse, J., Süß, T., Brinkmann, A.: Mercury: A transparent guided I/O framework for high performance I/O stacks. In: Proceedings of the 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP'17, St. Petersburg, Russia, March 06-08, 2017. pp. 46–53. IEEE Computer Society (2017), DOI: 10.1109/PDP.2017.83

21. Kunkel, J., Ludwig, T.: Performance evaluation of the PVFS2 architecture. In: Proceedings of the 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing, PDP'07, Washington, DC, USA, February 7-9, 2007. pp. 509–516. IEEE Computer Society (2007), DOI: 10.1109/PDP.2007.65

22. Dennis, J.M., Edwards, J., Loy, R., Jacob, R., Mirin, A.A., Craig, A.P., Vertenstein, M.: An application-level parallel I/O library for earth system models. International Journal of High Performance Computing Applications 26(1), 43–53 (2012), DOI: 10.1177/1094342011428143

23. Miller, E.L., Katz, R.H.: Input/output behavior of supercomputing applications. In: Proceedings of the 1991 ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis, SC'91, Albuquerque, NM, USA, November 18-22, 1991. pp. 567–576. IEEE Computer Society (1991), DOI: 10.1145/125826.126133

24. Dillow, D.A., Fuller, D., Wang, F., Oral, H.S., Zhang, Z., Hill, J.J., Shipman, G.M.: Lessons learned in deploying the worlds largest scale Lustre file system. Tech. rep., Oak Ridge National Laboratory (ORNL); Center for Computational Sciences (2010)

25. Smirni, E., Reed, D.: Workload characterization of input/output intensive parallel applications. In: Marie, R., Plateau, B., Calzarossa, M., Rubino, G. (eds.) Computer Performance Evaluation Modelling Techniques and Tools, Lecture Notes in Computer Science, vol. 1245, pp. 169–180. Springer Berlin Heidelberg (1997), DOI: 10.1007/BFb0022205

26. Kunkel, J., Ludwig, T.: Bottleneck detection in parallel file systems with trace-based performance monitoring. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008 – Parallel Processing, Lecture Notes in Computer Science, vol. 5168, pp. 212–221. Springer Berlin Heidelberg (2008), DOI: 10.1007/978-3-540-85451-7_23

27. Zimmer, M., Kunkel, J., Ludwig, T.: Towards self-optimization in HPC I/O. In: Kunkel, J., Ludwig, T., Meuer, H. (eds.) Supercomputing, Lecture Notes in Computer Science, vol. 7905, pp. 422–434. Springer Berlin Heidelberg (2013), DOI: 10.1007/978-3-642-38750-0_32

28. Carter, J., Borrill, J., Oliker, L.: Performance characteristics of a cosmology package on leading HPC architectures. In: Bougé, L., Prasanna, V. (eds.) High Performance Computing - HiPC 2004, Lecture Notes in Computer Science, vol. 3296, pp. 176–188. Springer Berlin Heidelberg (2005), DOI: 10.1007/978-3-540-30474-6_23

29. Dorier, M., Antoniu, G., Ross, R., Kimpe, D., Ibrahim, S.: Calciom: Mitigating I/O interference in hpc systems through cross-application coordination. In: Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS, Phoenix, AZ, USA, May 19-23, 2014. IEEE Computer Society (2014), DOI: 10.1109/IPDPS.2014.27

30. Fryxell, B., Olson, K., Ricker, P., Timmes, F.X., Zingale, M., Lamb, D.Q., MacNeice, P., Rosner, R., Truran, J.W., Tufo, H.: FLASH: an adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. The Astrophysical Journal Supplement Series 131(1), 273 (2000), DOI: 10.1086/317361

31. Skinner, D., Kramer, W.: Understanding the causes of performance variability in HPC workloads. In: Proceedings of the IEEE International Workload Characterization Symposium, Austin, TX, USA, October 06-08, 2005. pp. 137–149. IEEE Computer Society (2005), DOI: 10.1109/IISWC.2005.1526010

32. Uselton, A., Howison, M., Wright, N., Skinner, D., Keen, N., Shalf, J., Karavanic, K., Oliker, L.: Parallel I/O performance: From events to ensembles. In: Proceedings of the IEEE International Symposium on Parallel Distributed Processing, IPDPS, Atlanta, GA, USA, April 19-23, 2010. pp. 1–11. IEEE Computer Society (2010), DOI: 10.1109/IPDPS.2010.5470424

33. Hurrell, J.W., Holland, M., Gent, P., Ghan, S., Kay, J.E., Kushner, P., Lamarque, J.F., Large, W., Lawrence, D., Lindsay, K., et al.: The Community Earth System Model: A Framework for Collaborative Research. Bulletin of the American Meteorological Society 94(9), 1339–1360 (2013), DOI: 10.1175/BAMS-D-12-00121.1

34. Borrill, J., Oliker, L., Shalf, J., Shan, H.: Investigation of leading HPC I/O performance using a scientific-application derived benchmark. In: Proceedings of the ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis SC'07, Reno, NV, USA, November 10-16, 2007. pp. 1–12. IEEE Computer Society (2007), DOI: 10.1145/1362622.1362636

35. Bhatele, A., Mohror, K., Langer, S.H., Isaacs, K.E.: There goes the neighborhood: performance degradation due to nearby jobs. In: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '13, Denver, CO, USA, November 17-22, 2013. IEEE Computer Society (2013), DOI: 10.1145/2503210.2503247

36. Lofstead, J., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K., Wolf, M.: Managing variability in the IO performance of petascale storage systems. In: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10), New Orleans, LA, USA, November 13-19, 2010. pp. 1–12. IEEE Computer Society (2010), DOI: 10.1109/SC.2010.32

37. Lofstead, J., Polte, M., Gibson, G., Klasky, S., Schwan, K., Oldfield, R., Wolf, M., Liu, Q.: Six degrees of scientific data: Reading patterns for extreme scale science IO. In:

Proceedings of the 20th International Symposium on High Performance Distributed Computing, HPDC'11, San Jose, California, USA, June 08-11, 2011. pp. 49–60. ACM (2011), DOI: 10.1145/1996130.1996139

38. Wiedemann, M., Kunkel, J., Zimmer, M., Ludwig, T., Resch, M., Bönisch, T., Wang, X., Chut, A., Aguilera, A., Nagel, W., Kluge, M., Mickler, H.: Towards I/O analysis of HPC systems and a generic architecture to collect access patterns. Computer Science - Research and Development 28(2-3), 241–251 (2013), DOI: 10.1007/s00450-012-0221-5

39. Nieuwejaar, N., Kotz, D., Purakayastha, A., Ellis, C., Best, M.: File-access characteristics of parallel scientific workloads. IEEE Transactions on Parallel and Distributed Systems 7(10), 1075–1089 (October 1996), DOI: 10.1109/71.539739

40. Laboratory, E.O.L.B.N.: Global cloud resolving model simulations, ernest orlando lawrence berkeley national laboratory. `http://vis.lbl.gov/Vignettes/Incite19` (2014), accessed: 2014-08-11