How File Access Patterns Influence Interference Among Cluster Applications

Chih-Song Kuo*, Aamer Shah*†, Akihiro Nomura‡, Satoshi Matsuoka‡ and Felix Wolf*†

*RWTH Aachen University, Department of Computer Science, Aachen, Germany
{chih-song.kuo, wolf}@rwth-aachen.de

† German Research School for Simulation Sciences, Laboratory for Parallel Programming, Aachen, Germany
a.shah@grs-sim.de

‡ Tokyo Institute of Technology, Global Scientific Information and Computing Center, Tokyo, Japan
nomura.a.ac@m.titech.ac.jp, matsu@is.titech.ac.jp

***Abstract*—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. However, the degree of interference depends on the specific file access pattern. On the basis of synchronized time-slice profiles, we compare the interference potential of different file access patterns. We consider both micro-benchmarks, to study the effects of certain patterns in isolation, and realistic applications to gauge the severity of such interference under production conditions. In particular, we found that writing large files simultaneously with small files can slow down the latter at small chunk sizes but the former at larger chunk sizes. We further show that such effects can seriously affect the runtime of real applications—up to a factor of five in one instance. In the future, both our insights and profiling techniques can be used to automatically classify the interference potential between applications and to adjust scheduling decisions accordingly.**

I. INTRODUCTION

The computational demand of HPC applications is continuously growing, raising the performance expectations of cluster users to unprecedented levels. To accommodate such demands, HPC systems frequently employ specialized designs such as multi-dimensional torus networks, GPGPU-based accelerators, and powerful parallel file systems. The latter are needed to service 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 [1].

Such HPC applications that perform frequent or massive file access requests are also common. Examples include data-intensive codes such as the cosmic microwave background analyzer MADCAP [2] and the global cloud system resolving model GCRM [3]. They both write massive amounts of data during execution, resulting in large write requests. In contrast, the continuum mechanics solver OpenFOAM [4] and Community Atmosphere Model (CAM) [5] of the Community Earth System Model (CESM) [6] frequently checkpoint their state, resulting in small but recurring writes. Overall, very different classes of file-access patterns can be distinguished. These patterns differ not only in how they access the file system themselves but also in their sensitivity to simultaneous file accesses by other applications. Furthermore, they generate different levels of interference for other I/O-intensive applications, making access patterns an important factor in the degree of contention for file system resources.

File system contention and the associated performance degradation are well known [7]. File-access patterns, specifically read patterns [8], and the effects of request size and application scale [9], have also been studied before in this context from a single-application perspective. 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 benchmark capable of producing three distinct file access patterns, simulating those of real applications. Two of these patterns mimic application checkpointing and small data accesses, while the third pattern mimics large file writes. We explored the interference potential of these patterns by running them simultaneously against each other, either in the form of micro-benchmarks or realistic applications covering checkpoint-intensive and data-intensive access patterns. We not only observed different levels of interference between different patterns, but also saw some consistent behaviors such as data-intensive I/O dominating checkpointing at smaller write-request sizes, with the trend being reversed for larger write-request sizes. 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 [10], 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. We first provide the necessary background information on parallel file systems in Section II. In Section III, we then 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 monitoring added to LWM^2 for the purpose of this study. After that, we present our results in Section IV, ranging from benchmark-only experiments to measurements with realistic applications. Finally, we review related work in Section V before we draw our conclusions and outline future perspectives in Section VI.

II. Parallel file system

To accommodate an increasing number of concurrent file accesses, cluster file systems evolved from a simple client-server model in the style of NFS to 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 depicted in Figure 1. Clients running on compute nodes forward file-access requests to the I/O servers. The 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 have been developed and publicly released. Below, we describe in more detail two popular parallel file systems we used in our experiments.

A. Lustre

Lustre is a file-storage system for clusters used by many of the Top500 HPC systems [11]. It offers up to petabytes of storage capacity and provides up to gigabytes per second of I/O throughput. Its architecture distinguishes two basic types of servers: *metadata servers* (MDSs) and *object storage servers* (OSSs). 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 seeks 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.

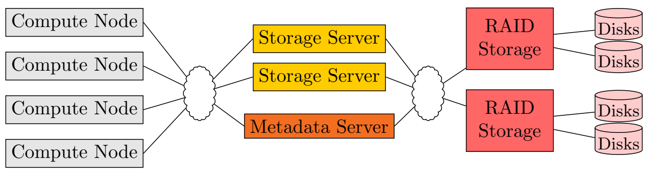


Fig. 1: Structure of a parallel file system with a separate metadata server.

B. GPFS

The General Parallel File System (GPFS) is a proprietary parallel file system developed by IBM [12]. It is often found on Blue Gene systems but is also available on other HPC clusters such as TSUBAME. 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.

III. Approach

Many HPC applications perform extensive I/O operations. They employ different I/O libraries and file formats and create different process-to-file ratios. Because a significant proportion of applications still use POSIX-IO or MPI-IO in a classic one-file-per-process manner [13], 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 [2], at least on our test systems and on many others, we restrict ourselves to MPI-IO in this study.

A. I/O access patterns

HPC applications exhibit a variety of file access patterns, of which frequent checkpointing and writing of large output files are considered here. We implemented three characteristic patterns corresponding to these two 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.

1) Open-write-close: The first pattern we consider is called open-write-close (OWC) (Figure 2a). In this pattern, each process opens a file, writes data to it and then closes it. The pattern is commonly used for checkpointing in many applications, such as OpenFOAM [4], CESM [6] and Flash [14]. This access pattern generates a large number of metadata operations while the actual amount of data written to files is comparatively small. On systems with limited metadata resources such patterns can quickly create a bottleneck at scale.

2) Write-seek: In the write-seek (WS) pattern (Figure 2b), 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 and performs massive small file accesses. However, it generates less metadata traffic as the file is continuously kept open. Between individual writes, only seek operations take place.

```
procedure OPEN-WRITE-CLOSE
    loop
        File Open
        Write $chunksize$
        Flush I/O Writes
        File Close
    end loop
end procedure
```

(a) Open-Write-Close

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

(b) Write-Seek

```
procedure AGGREGATE-WRITE
    File Open
    loop
        Write $chunksize$
    end loop
    File Close
end procedure
```

(c) Aggregate-Write

Fig. 2: Three I/O access patterns.

3) Aggregate-write: In the aggregate-write (AW) pattern (Figure 2c), a process opens a file at the beginning and then continues to append chunks of data to it. The file is closed at the end of execution. This pattern is similar to large writes in applications such as MADCAP [2] and GCRM [3]. The pattern involves 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.

Careful investigation of the client-side I/O caching behavior created the necessity of flushing 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 Lustre client software and get overwritten with the next write. This issue does not occur for writes of large chunks as the chunk size is larger than the OST buffer size. This issue also does not effect aggregate-write, in which small writes are aggregated to the OST buffer and eventually committed to the file system. To have a consistent benchmark, writes were flushed for both Lustre and GPFS, and for all chunk sizes.

B. 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 how interference effects evolve over the runtime of a specific, more complex probe application, we let the signal benchmark also produce its pattern in a *periodic* fashion, with the I/O activity being interrupted by phases of no I/O activity. Whenever the signal shows activity, the probe may suffer a dent whose depth indicates the severity of the interference.

To measure how the I/O throughput of an application changes, we use the profiler LWM^2 [10] 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 for our study is LWM^2's ability to represent performance dynamics in *time slices*. In addition to producing 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 very 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.

To be able to keep our measurements as clean as possible from these two effects, we extended LWM^2 to monitor also the activities of the I/O server during execution of an application. The server activities are captured every time slice, allowing us to correlate events across applications and I/O servers. In particular, it allows us to filter out runs where file server load is 10% larger than the application I/O traffic captured by POSIX/MPI I/O wrappers. In addition, the 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 III-C. 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.

C. Server-side imbalance

In some experiments, we observed severe imbalance among the processes of an application that occurred sporadically on both file systems. In such cases, most processes finished within the expected time, while the remaining ones had to keep

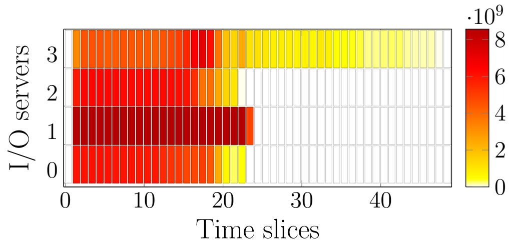


Fig. 3: Server side imbalance when writing files (GPFS). The darker the color, the higher the throughput in a time slice.

performing I/O for a significantly longer duration, sometimes more than twice as long. One major factor revealed in a closer investigation of the imbalance effect was unbalanced load on the file-server side, as shown in Figure 3. In particular, with Lustre, files were randomly assigned to an OST in 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%. 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 occured to a lesser extent with large files because they were automatically striped over all NSDs, but more predominantly with small files below the stripe size presumably due to the same reason. Besides the OST/NSD load-imbalance, other factors, such as the straggler phenomenon [15], 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 affects only the later stage of an run, in which 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.

IV. 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. We then confirmed our findings by executing the benchmarks against two realistic applications, OpenFOAM and MADbench2, used for fluid dynamics and cosmic simulations, respectively. Finally, we analyzed the interference effect between two instances of these realistic applications.

A. Environment

The results were obtained on the TSUBAME2.5 supercomputer [16] 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, 2.93GHz) 6-core processors, three NVIDIA Tesla K20X (GK110) GPUs, and 58GiB 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. It was ranked 11th in the Top500 list of November 2013.

PFS	Mount point	Metadata server	File server	NSDs/OSTs	Throughput
GPFS	/data0	N/A	4	14/server	20GB/s
Lustre	/work1	1 (+1 standby)	8	13/server	50GB/s

TABLE I: Specifications of file systems on TSUBAME used in our experiments.

TSUBAME offers GPFS and Lustre file systems for parallel I/O at different mount points. The GPFS on /data0 is hosted on four file servers (NSD servers), each connected to 14 RAID storage devices (NSDs), while the 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 is set to 16%, meaning that storages are selected mostly in a round robin fashion. Additionally, TSUBAME also provides 120GB 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 I provides a summary of the two file systems on the mount points we used.

For validation purposes, we also re-ran a limited set of our experiments on JUROPA, a 2208-node cluster at Jülich Supercomputing Centre, Germany, and on a 48-node cluster at the Fujitsu HPC Benchmark Center, Paderborn, Germany. We were able to completely reserve the later system, allowing us to run our experiments in a noise-free fully-controlled environment. Both of the systems only had Lustre file systems available with different I/O server setups. Nonetheless, our observations regarding interference trends for the three patterns on all the systems were consistent.

B. Experiment configuration

In our experiments on TSUBAME, a single instance of an application consisted of 256 processes, utilizing 64 compute nodes. As the experiments were carried out on a production system, care was taken to filter out runs with more than 10% external noise. The filtering was done by using the I/O server monitoring module of LWM^2. We also repeated each experiment five times and took the best-performing, least-interfered run.

We executed the patterns with file sizes ranging from 1MiB to 128MiB 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. At the end of the execution, the resulting files for aggregate-write pattern had accumulated up to 2GiB per process in the one-file-per-process mode, or up to 512GiB in the single-shared-file mode.

C. *Micro-benchmarks*

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 benchmark once as a signal and once as a probe, resulting in a total number of nine scenarios (i.e., $\{OWC, WS, AW\}^2$). The severity of the interference effect was then quantified in terms of the percentage degradation in throughput T, defined as:

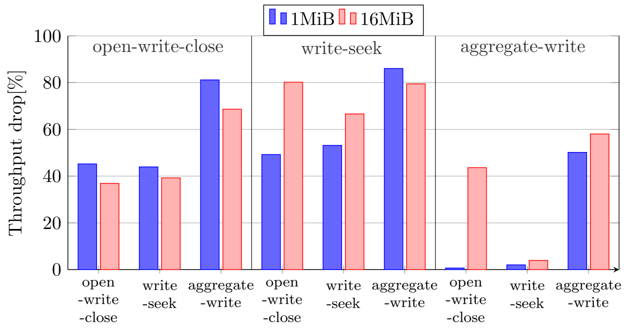
$$T = \frac{T_{standalone} - T_{interfered}}{T_{standalone}} \times 100$$

A high value of the degradation indicates severe interference inflicted by the signal pattern. We executed the complete set of combinations on both GPFS and Lustre for chunk sizes of 1MiB and 16MiB.

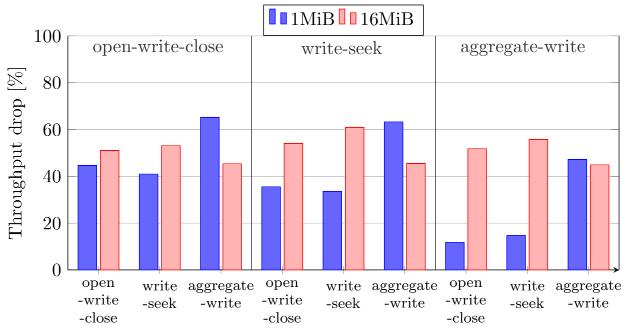
1) GPFS: Figure 4a shows the throughput drop observed with all pattern combinations, for chunk sizes of both 1MiB and 16MiB. With the smaller chunk size, we found aggregate-write to have a clearly higher interference potential than the other two patterns. When open-write-close and write-seek are executed against each other, their throughput drops by about 50%. This can be explained by 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 1MiB, occupying most of the I/O resources. At a chunk size of 16MiB, 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 1MiB case, while aggregate-write is more affected by the other two patterns. However, even at a chunk size of 16 MiB, write-seek still shows less active interference potential and is influenced more by other patterns.

As chunk size seem to be a crucial factor in the interference potential of the above patterns, we investigated this more closely by running open-write-close and write-seek against aggregate-write for chunk sizes ranging from 1MiB to 128MiB on a logarithmic scale. The results are shown in Figure 5a. For open-write-close, there is a clear trend for I/O resources to be shared more evenly among the two patterns as the chunk size increases, with 64MiB being more or less the break-even point. Beyond this point, open-write-close starts to dominate, thus degrading aggregate-write more than it suffers throughput reduction itself. Write-seek shows a similar trend but with a shifted slope. Only after a 32MiB chunk size does write-seek with interfere aggregate-write significantly. 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 128MiB aggregate-write still dominates. If the trend continued, and if it was similar to open-write-close, write-seek would start to dominate aggregate-write at larger chunk sizes.

2) Lustre: We repeated the same set of experiments on Lustre. The results from the nine pair-wise combinations of patterns for 1MiB and 16MiB are shown in Figure 4b. The general trend of the interference potential for the two chunk



(a) GPFS



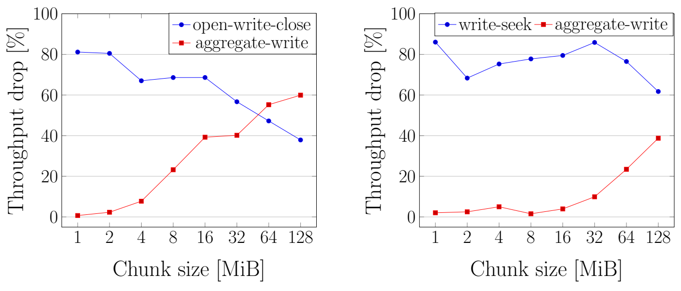
(b) Lustre

Fig. 4: Throughput drop when the patterns are executed against each other. A higher bar means less throughput and higher passive interference. The top patterns indicate the probe, while the patterns below the x-axis indicate the signal.

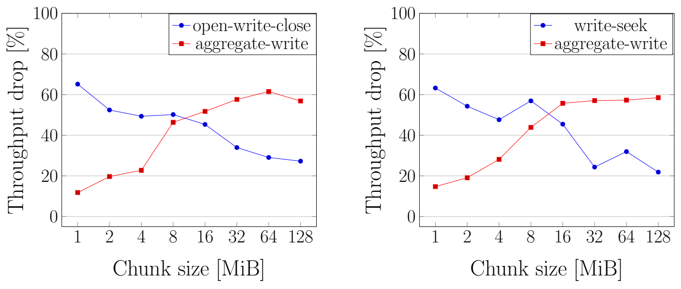
sizes is the same as on GPFS but with different intensities. At a chunk size of 1MiB, aggregate-write again generates most of the interference, while being itself the least affected one. However, the disparity is not as strong as on GPFS. At a chunk size of 16MiB, the interference potential of all the patterns is roughly the same.

We also explored the sensitivity of interference to chunk sizes by running open-write-close and write-seek against aggregate-write for a range of chunk sizes from 1MiB to 128MiB on a logarithmic scale. The results are summarized in Figure 5b. The trends of both patterns against aggregate-write are similar to each other, and also have similarities to the trends on GPFS but with aggregate-write being dominated earlier. Open-write-close and write-seek are affected more by aggregate-write at smaller chunk sizes but begin to dominate as the chunk size increases. The break-even point is between 8MiB and 16MiB, beyond which aggregate-write is increasingly dominated. However, the slopes start to flatten for very large chunk sizes. As a comparison, the interference potential of write-seek on Lustre is higher compared to GPFS, as write-seek starts to dominate aggregate-write beyond 16MiB.

3) Discussion: From the above results it is clear that different I/O access patterns show different interference potentials. The chunk size is also an important factor in determining which pattern is dominated. At smaller chunk sizes, large writes (aggregate-write) prevails over frequent small writes (i.e., open-write-close and write-seek), causing a notable degradation of throughput for the latter while showing little impact on the former. However, as the chunk size increases,



(a) GPFS



(b) Lustre

Fig. 5: Drop in throughput suffered by open-write-close and write-seek when executed against aggregate-write with different chunk sizes. A higher value means less throughput and higher passive interference.

the balance is shifted in favor of the smaller writes. At a certain point, both types of behaviors suffer equally, beyond which the performance of large writes drops to a greater degree. On Lustre, open-write-close and write-seek show similar degradation trends, while on GPFS, write-seek 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 to interference. At least, this would explain the trend reversal shown in Figure 5b. As the chunk size, and with it the number of different blocks written by aggregate-write, increases, the density of metadata operations shrinks.

4) Shared files: 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.

We present results for Lustre in Figure 6. In our experiments, we observed that aggregate-write dominated the other two patterns at 1 MiB chunk size, but showed reduced interference potential at a chunk size of 16 MiB, where the interference potential of all patterns seems balanced. Both of these observations are consistent with the one-file-per-process case. Only when running aggregate-write against itself at a chunk size of 1 MiB did we observe much less interference than with one file per process. Nevertheless, increasing the chunk size of this pattern combination to 16 MiB blends it seamlessly into the balanced overall picture.

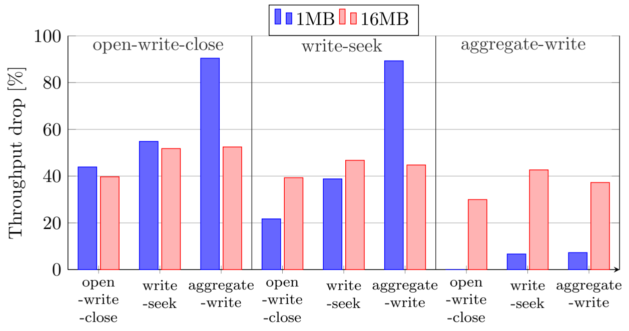


Fig. 6: Shared-file throughput drop on Lustre when the patterns are executed against each other.

The interference potentials followed the same general trend on GPFS. Aggregate-write dominated the other two patterns significantly at a chunk size of 1 MiB. While at 16 MiB, even though it was still dominant, aggregate-write generated 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 pair-wise execution of the patterns, in which one instance would completely dominate the other instance, effectively serializing the I/O traffic between the pairs. This near serialization of the I/O traffic was seen both when running patterns against themselves and against different patterns. 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.

D. MADbench2

MADbench2 is derived from the cosmic microwave background radiation analysis software MADCAP. MADbench2 performs dense linear algebra calculation using ScaLAPACK [17]. It has very large memory demands and its required matrices generally do not fit in memory. As a result, the calculated matrices are written to disk and re-read when required. During a normal execution, MADbench2 writes large amounts of data with periodic seeks in between, making its behavior similar to write-seek. Due to its I/O intense nature, it has been used to benchmark I/O performance of HPC systems [2]. For our experiments, we configured MADbench2 to run in I/O-mode. In this mode, MADbench2 replaces all computations with busy-waits, while still maintaining the I/O behavior of the actual software. We further configured MADbench2 to use MPI-IO with one file per process and ran it against all the three patterns. MADbench2 was executed on both GPFS and Lustre with chunk sizes of 1 MiB and 16 MiB. To more accurately gauge the throughput drop values in the experiments, we only considered the time slices in which file writes were performed.

1) GPFS: At a chunk size of 1 MiB the interference of aggregate-write was so strong that some of the runs were not completed in an acceptable time, which is why we cannot calculate the throughput drop. But the runtime dilation suggests that aggregate-write was the biggest source of interference among the three patterns. At a chunk size of 16 MiB, aggregate-write slowed the throughput of MADbench2 slightly more than the other two patterns, similar to our micro-benchmark-only results in the three rightmost column pairs of Figure 4a.

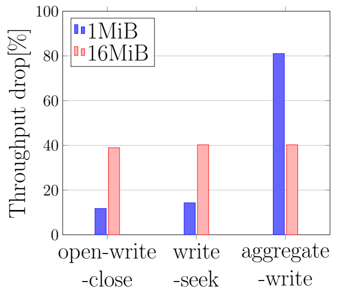


Fig. 7: Throughput drop when MADbench2 is executed concurrently with the three patterns on Lustre.

2) Lustre: The results of the experiments on Lustre are shown in Figure 7. Again, similar to our micro-benchmark-only results, at 1 MiB aggregate-write has a much stronger effect than the other patterns, a difference that almost disappears when the chunk size is increased to 16 MiB.

E. OpenFOAM

OpenFOAM [4], 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 modularized 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. However, as OpenFOAM regularly opens and closes files as a consequence of frequent checkpointing, we used the file close rate as a substitute for the throughput rate. 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.

For our experiment, we ran the cavity example from the official tutorial. 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 patterns. We set the chunk size of the patterns to 1MiB, the smallest chunk size from our experiments with the micro-benchmark. We executed the runs on both GPFS and Lustre, and adjusted the runtime of the patterns to fully overlap with OpenFOAM's execution.

1) GPFS: OpenFOAM experienced degraded I/O performance when executed concurrently with all the three patterns, leading to degraded I/O performance. The throughput drop caused by each of the patterns is shown in Figure 8. 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 to more than 90% drop in throughput. The other two patterns of open-write-close and write-seek also generated interference, but to a smaller degree, degrading OpenFOAM's throughput by 55%. These observations are similar to our pattern vs. pattern experiments, where we found that aggregate-write dominated at small chunk sizes.

To further understand the I/O interference dynamics during concurrent execution, we executed OpenFOAM against periodic modes of open-write-close and aggregate-write. In this mode, the benchmark's I/O access phases alternate with silence. This periodic mode highlights the effects of interference during the I/O access phases. Figure 9a and Figure 9b 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% in I/O access phases of the pattern, compared to the no I/O access phases. On the other hand, against aggregate-write, OpenFOAM degrades by up to 95% when the pattern performs I/O accesses. This is clearly visible as the file close rate of OpenFOAM adopts a periodic behavior under interference.

2) Lustre: On Lustre, we observed a similar degradation trend when OpenFOAM was executed against the three patterns, as shown in Figure 8. OpenFOAM experienced degradation from all three patterns, but to a lesser extent from the open-write-close and write-seek patterns, degrading the throughput by about 40% and 30% respectively. Aggregate-write, however, generated severe interference and degraded the throughput by more than 80%. In contrast, the I/O throughput of the aggregate-write pattern did not suffer from significant degradation when run against OpenFOAM. This is again consistent with our previous observation for small chunk sizes in the pure micro-benchmark comparison.

F. Inter-application interference

With our knowledge of how isolated access patterns interfere with realistic applications, we also investigated the interference between realistic applications, as can occur in a live production system. For this purpose, we ran MADBench2 and OpenFOAM first against themselves and later against each other. At the process counts used in our experiments, the two applications have a lower I/O intensity than our benchmarks. As a result, some runs on Lustre showed no interference. The

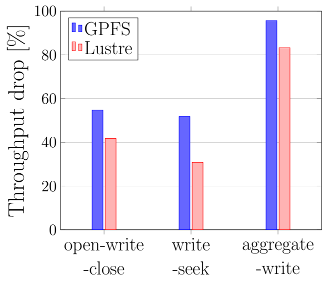
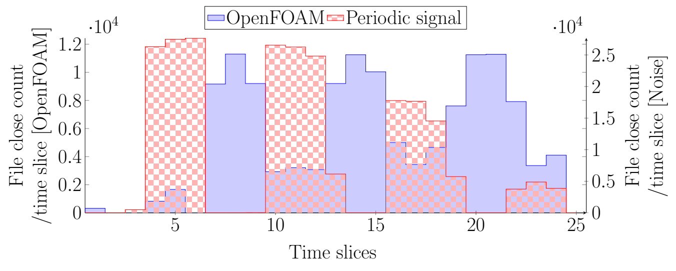
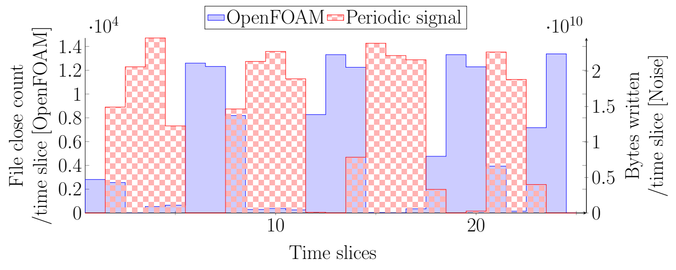


Fig. 8: Throughput drop of OpenFOAM vs. all three patterns on GPFS and Lustre with a chunk size of 1 MiB.



(a) OpenFOAM vs. open-close



(b) OpenFOAM vs. aggregate-write

Fig. 9: Time-slice view of OpenFOAM when executed concurrently with two patterns on GPFS.

higher peak bandwidth of the Lustre I/O subsystem, as shown in Table I, offers an explanation.

Executing two instances of MADbench2 concurrently resulted in 20% runtime dilation on GPFS, while producing negligible interference on Lustre. Executing two instances of OpenFOAM concurrently showed 20% and 25% runtime dilation on Lustre and GPFS, respectively. Because our first attempts to run MADbench2 vs. OpenFOAM on Lustre only resulted in runtime dilation comparable to the run-to-run variation, we optimized MADbench2 to achieve higher throughput according to recommendations in literature [2] and then executed ten runs on Lustre. MADbench2 itself experienced negligible interference while OpenFOAM's runtime was dilated by 10%. Although these effects are smaller in magnitude than those in our benchmark studies, the results confirm our general observation that at this small chunk size data-intensive I/O has a stronger effect on checkpoint-intensive I/O than vice versa. Due to high run-to-run variation when running MADbench2 vs. OpenFOAM on GPFS, experiments on this file system are still ongoing.

V. Related Work

Inter-application interference can significantly reduce performance [10], [18]. In particular, concurrent access to parallel file systems leads to contention and performance degradation [1], especially at scale [19]. Such degradation has been studied from a single application's perspective [2], [20] and from a complete system's perspective using a single application [21], [22]. The influence of an application's process count on the degree of inter-application interference at the level of the I/O subsystem has also been investigated [23]. In our work, we focus on the interference of simultaneous I/O requests issued by multiple applications, with special emphasis given to their file access patterns.

Interference on parallel file systems can be reduced by separating I/O requests either in space or time. The latter means avoiding co-scheduling of resources while the former includes I/O-hardware level partitioning through I/O zoning—potentially at the expense of reduced aggregate I/O bandwidth [24], [25]. Burst buffers that reduce peak I/O traffic include elements of both space and time partitioning [26].

Several authors have also characterized HPC applications based on their file access patterns [26]. Specifically, Lofstead et al. [8] analyzed read access patterns with NetCDF and HDF5 file formats, concentrating on end-to-end I/O performance of a single application. Aggregate I/O throughput for different I/O request sizes and client and server counts has been surveyed for the PVFS2 file system, analyzing one behavior at a time [9]. Here, we show how different write-access patterns of concurrently running applications interfere.

Several tools offer functionality to analyze the performance of file I/O. Darshan [13] is a performance tool that captures the I/O profile of an application. It has also been deployed system-wide on a Blue Gene system to characterize I/O loads of various applications. IPM-IO is an extension of the lightweight profiler IPM with I/O tracing capabilities [22]. PIOviz profiles MPI applications while also tracing PVFS2 server activities [27]. We use LWM^2 for our analysis, utilizing its unique feature of globally synchronized time-slice profiling to capture application interference. For this particular analysis, we extended its functionality to monitor the I/O server load alongside application requests. Finally, the I/O subsystem analysis tool SIOX [28], [29] identifies bottlenecks along the complete I/O path and proposes application optimizations, taking their access patterns into account. Our LWM^2-based infrastructure is designed to complement such efforts, adding a cross-application pattern optimization facet.

VI. Conclusion

We analyzed different inter-application interference effects caused by the interaction between various I/O access patterns, classified by their behavior, write chunk size, and sharing mode. Specifically, we found that at small chunk sizes data-intensive applications may significantly slow down checkpointing-intensive applications, 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 increasingly reversed as the chunk size is increased.

Given the shared nature of most parallel file systems, preventing I/O interference in its entirety 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, to make such a separation successful it is important to decide what traffic should be separated. Leveraging techniques now demonstrated 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.

To support future interference-aware file-system designs, we plan to further extend LWM^2 to recognize application I/O access patterns automatically and suggest 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.

VII. Acknowledgments

This research was supported by JST, CREST (Research Area: Advanced Core Technologies for Big Data Integration) and the G8 Research Councils Initiative on Multilateral Research, Interdisciplinary Program on Application Software towards Exascale Computing for Global Scale Issues. Moreover, we would like to thank Hitoshi Sato and Jens Domke for their advice regarding parallel I/O system and I/O networks. Finally, we would like to acknowledge the support of ict GmbH, which gave us access to the Fujitsu HPC Benchmark Center.

References

[1] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "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*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–12.

[2] J. Borrill, L. Oliker, J. Shalf, and H. Shan, "Investigation of leading HPC I/O performance using a scientific-application derived benchmark," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis.*, ser. SC '07. Washington, DC, USA: ACM, 2007, pp. 1–12.

[3] Ernest Orlando Lawrence Berkeley National Laboratory, "Global cloud resolving model simulations," http://vis.lbl.gov/Vignettes/Incite19.

[4] H. Jasak, A. Jemcov, and Z. Tukovic, "Openfoam: A C++ library for complex physics simulations," in *International workshop on coupled methods in numerical dynamics*, 2007, pp. 1–20.

[5] J. M. Dennis, J. Edwards, R. Loy, R. Jacob, A. A. Mirin, A. P. Craig, and M. Vertenstein, "An application-level parallel I/O library for earth system models," *International Journal of High Performance Computing Applications*, vol. 26, no. 1, pp. 43–53, 2012.

[6] J. W. Hurrell, M. Holland, P. Gent, S. Ghan, J. E. Kay, P. Kushner, J.-F. Lamarque, W. Large, D. Lawrence, K. Lindsay *et al.*, "The community earth system model." *Bulletin of the American Meteorological Society*, vol. 94, no. 9, 2013.

[7] National Institute for Computational Sciences, "I/O and Lustre usage," https://www.nics.tennessee.edu/computing-resources/file-systems/io-lustre-tips.

[8] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu, "Six degrees of scientific data: Reading patterns for extreme scale science IO," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 49–60.

[9] J. Kunkel and T. Ludwig, "Performance evaluation of the PVFS2 architecture," in *15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Feb 2007, pp. 509–516.

[10] A. Shah, F. Wolf, S. Zhumatiy, and V. Voevodin, "Capturing inter-application interference on clusters," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2013, pp. 1–5.

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

[12] IBM, "An introduction to GPFS version 3.5," http://www-03.ibm.com/systems/resources/introduction-to-gpfs-3-5.pdf.

[13] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," in *IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, May 2011, pp. 1–14.

[14] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo, "Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes," *The Astrophysical Journal Supplement Series*, vol. 131, no. 1, p. 273, 2000.

[15] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki, "Characterizing output bottlenecks in a supercomputer," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society, 2012.

[16] Tokyo Institute of Technology Global Scientific Information and Computing Center, "Tsubame computing services," http://tsubame.gsic.titech.ac.jp.

[17] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "Scalapack: a scalable linear algebra library for distributed memory concurrent computers," in *Frontiers of Massively Parallel Computation, Fourth Symposium on the*. IEEE, 1992, pp. 120–127.

[18] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "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*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 41:1–41:12.

[19] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 40:1–40:12.

[20] H. Shan and J. Shalf, "Using IOR to analyze the I/O performance for HPC platforms," in *Cray User Group Conference*, 2007.

[21] W. Yu, J. Vetter, and H. Oral, "Performance characterization and optimization of parallel I/O on the Cray XT," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2008, pp. 1–11.

[22] A. Uselton, M. Howison, N. Wright, D. Skinner, N. Keen, J. Shalf, K. Karavanic, and L. Oliker, "Parallel I/O performance: From events to ensembles," in *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–11.

[23] C.-S. Kuo, A. Nomura, S. Matsuoka, A. Shah, F. Wolf, and I. Zhukov, "Environment matters: How competition for I/O among applications degrades their performamce," in *IPSJ SIG Technical Report*, vol. 2013-HPC-142, no. 11, 2013, pp. 1 – 7.

[24] S. Sumimoto, "FEFS performance evaluation on K computer and Fujitsu's roadmap toward Lustre 2.x," in *Lustre File System User Group*, 2013.

[25] I. Kitayama, "A user's experience with FEFS," in *Lustre File System User Group*, 2013.

[26] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, 2012, pp. 1–11.

[27] J. M. Kunkel and T. Ludwig, "Bottleneck detection in parallel file systems with trace-based performance monitoring," in *Euro-Par – Parallel Processing*, ser. Lecture Notes in Computer Science, E. Luque, T. Margalef, and D. Benítez, Eds. Springer Berlin Heidelberg, 2008, vol. 5168, pp. 212–221.

[28] M. Wiedemann, J. Kunkel, M. Zimmer, T. Ludwig, M. Resch, T. Bönisch, X. Wang, A. Chut, A. Aguilera, W. Nagel, M. Kluge, and H. Mickler, "Towards I/O analysis of HPC systems and a generic architecture to collect access patterns," *Computer Science - Research and Development*, vol. 28, no. 2-3, pp. 241–251, 2013.

[29] M. Zimmer, J. Kunkel, and T. Ludwig, "Towards self-optimization in HPC I/O," in *Supercomputing*, ser. Lecture Notes in Computer Science, J. Kunkel, T. Ludwig, and H. Meuer, Eds. Springer Berlin Heidelberg, 2013, vol. 7905, pp. 422–434.