FiLiP: A File Lifecycle-based Profiler for hierarchical storage

Adrian Khelili, Sophie Robert and Soraya Zertal

Abstract—**The increasing gap between computing speed and storage latency leads to possible I/O bottlenecks on massively parallel computers. To mitigate this issue, hierarchical storage provides multi-tiered configurations where each tier has its own physical characteristics and associated performance. Selecting the most appropriate file placement policy on this multi-tiered storage is difficult and there is to our knowledge no tool that systematically provides statistics and metrics for optimal file policy selection. In this paper, we present FiLiP (File Lifecycle Profiler), a software which provides statistics and metrics for a better understanding of file access by applications and the consequences on file movements across hierarchical storage. After the description of FiLiP's main features and architecture, we highlight the usefulness of our tool using three I/O intensive simulation HPC applications: NEMO, S3DIO and NAMD and a three-tiered burst buffer.**

Index Terms—**I/O, File lifecycle, Profiling, Burst Buffer, Hierarchical storage, HPC**

I. Introduction

On modern High Performance Computing (HPC) systems, important efforts are put in improving massively parallel computations and communication between compute nodes in order to deliver a higher performance. However, the huge gap between the computing capacity and the storage system latency leads to I/O bottlenecks, and a similar effort has to be made on the I/O and storage systems as a response to the large amount of data generated, manipulated, shared and transferred by modern applications. Recent computer architectures provide hierarchical storage systems with different tiers [28], each with its own physical characteristics based on the device technology and its associated performance metrics, such as latency or throughput [8]. Because of the performance disparity between these tiers, a good file placement is required in order to make the most out of the performance of each one and improve applications performance. Understanding the storage hierarchy behavior, along with the file access performed by the application, is thus key to adapt the data placement to the application's access profile.

Such a file data placement policy can be considered effective when it increases the performance of the system by placing hot files in the best performing tiers and cold files in the worst performing tiers, so that the files with the highest probability of being read are quickly available to the application [8]. To distinguish between these two access and achieve an efficient data placement, the first step is to perform a thorough profiling of the application and its files manipulation, to characterize file re-use throughout the application's lifetime. From these observations, an optimal file placement policy can then be designed and selected. However, selecting the best policy is often a complicated task, as the literature is rich of different strategies: Random, LRU and LFU in their basic versions [13] [16] or optimized ones [10] [24], methods using tuning as ARC [21], methods introducing additional history information as LIRS [15] or methods combining tiering and caching [30]. Novel approaches attempting cache management using statistics of past requests [12] or through machine learning techniques as in [14] [31] [11] [4] have proven successful, by focusing on I/O patterns detection to predict which block should be loaded into the different cache tiers at a given time.

Among all these different approaches, the selection of the optimal placement should rely on objective criteria and metrics describing the placement, such as the cache hits and cache miss rates, file re-use rate, file lifecycles…To our knowledge, none of these metrics have been systematically included into an I/O profiler and correlated with the behavior of the application at the file level for selecting an optimal file placement, as the literature mainly focuses on the application level. We believe that combining the profiling at the file level with the description of file movements through hierarchical and heterogeneous storage tiers fills this literature gaps, and we suggest in this paper a new tool called FiLiP (**Fi**le **Li**feycle **P**rofiler) to provide data analysis at the file level, by allowing the visualization of operations performed on the file by the application, and systematically including statistics quantifying the quality of file placement policies in the hierarchical storage.

This paper is structured as follow. We present in section II the related works and point out the novelty of FiLiP compared to existing profiling software. In section III, we describe FiLiP's main features. Section IV presents its utilisation within three different HPC applications running on hierarchical storage, and present an analysis of their file-level manipulation that can only be done using our new suggested software. In section V, we conclude the paper by providing some insight on some works we are currently inquiring.

Adrian Khelili: Atos BDS R&D Data Management Li-PaRAD, UPSaclay-UVSQ, France. (E-mail: adrian.khelili@atos.net)

Sophie Robert: Atos BDS R&D Data Management Echirolles, France. (E-mail: sophie.robert@atos.net)

Soraya Zertal: Li-PaRAD, UPSaclay-UVSQ Guyancourt, France. (E-mail: soraya.zertal@uvsq.fr)

DOI: 10.36244/ICJ.2022.4.4

age, and present an analysis of their file-level manipulation that can only be done using our new suggested software. In section V, we conclude the paper by providing some insight on some works we are currently inquiring.

II. Related works

Profiling has been used in different areas and at different levels from the top of the software stack to the hardware level. This technique has proven its efficiency and accuracy to investigate any component and understand its behavior when submitted to various execution conditions. The literature is rich of many profiling tools dedicated to power and energy, computing, memory and storage hierarchies. For example, in the energy and power domain to understand energy consumption by providing a suite of statistics [17] [26] and experimental methodology [29]. When it comes to tasks placement, synchronization and communication, monitoring and profiling tools such as [9] [25] for MPI and MapReduce allow to understand applications running on massively parallel architectures. At the memory hierarchy level, profiling guides heap management [22] and improve the suitability of mapping parallel applications [7].

At the storage hierarchy level, several I/O profilers exist and aim to understand applications I/Os such as IOPIN [18] and IOPRO [19]. Their principal utility is to identify I/O bottlenecks and thus better exploit the system potential. One of the most popular I/O profiler is Darshan [5] : a characterization tool developed at Argonne National Lab and designed to capture an accurate picture of the I/O behavior of an application to help developers tune their application parameters by measuring the effect of a parameter set on execution time. Rather than capturing a complete trace of each I/O, Darshan characterizes the job by recording global statistics, such as counters for POSIX operations and their timestamps, cumulative bytes read and written, for a compact representation in memory. A similar profiling tool is Atos IO Instrumention tool (IOI) [2] with a web interface delivering a maximum of information to users by providing statistics as time series describing a wide range of I/O related statistics.

All these profilers are application-oriented and thus have only one perspective when profiling and analyzing the system. We propose through FiLiP the possibility to investigate and analyze the system from the file perspective, by providing the novel features that:

- **Enables profiling at the file-level**: FiLiP provides the possibility to characterize the I/O behavior of an application at the file level for a thorough understanding of file lifecycles and data movements through hierarchical storage. It displays metrics at the application level as well, such as hit/miss rates, total reads, writes, to provide a higher level description of the application's I/O behavior.
- **Provides a standardized interface**: FiLiP relies on a suggested standardized format for description of file movement between the storage hierarchy tiers that can be implemented with any monitoring system.
- **Evaluates the quality of a destaging policy**: Given a destaging policy described through this standardized format, FiLiP evaluates its quality through relevant statistics. It acts then as a performance evaluation tool for new tiers-management strategies.

III. Features and architecture

A. Definitions and notations

We define the following terms that will be referred to throughout the rest of the paper:

- **Storage tiers**: n storage tiers, ordered hierarchically depending on their access latency (for example, $tier_1$ = RAM, $tier_2$ = NVMe, $tier_3$ = HDD). We will denote $tier_n$ the tier at level n.
- **File lifecycles**: a temporal series of operation performed on the file $(o_t)_{1 \leq t \leq T}$, with o_t an operation in the POSIX set $\{$READ, WRITE, OPEN, CLOSE$\}$.
- **File placement policy**: a temporal series describing the data movement from and to a storage tier (m_{ij}) with i the source tier and j the target one.

B. Main features

FiLiP gives the user an easy to access Web interface, providing:

- **A general understanding of the file manipulation behavior of the application**: Visualization of general statistics on the file behavior during the application's execution, as can be seen in figure 4.
- **The visualization of file lifecycles**: Visualization of file lifecycles to observe how the application manipulates its different files during their lifespans through POSIX operations, as can be seen in figure 4.
- **The visualization of file placement policies**: Files movements are correlated with the file lifecycles to assess the impact of file accesses on moves across the hierarchical storage. File placement policies can also be characterized through generic statistics for proper evaluation as depicted in figure 1 and described in section III-C.
- **The visualization of operations per file**: These statistics are displayed as a table that contains for each accessed file the number of *read*, *write*, *open*, and *close* operations, as well as the volume of data manipulated per file for each of read and write operations in Gb.
- **Visualisation of tiers usage**: FiLiP's provide the usage percentage for each tier, at each moment of the application execution. The fill rate of the tiers are displayed as a time series, that offers a global view about fluctuations in tiers usage during the execution.

C. Evaluation metrics

As displayed in figure 1, FiLiP's main menu provides the following metrics to evaluate the quality of file placement policies:

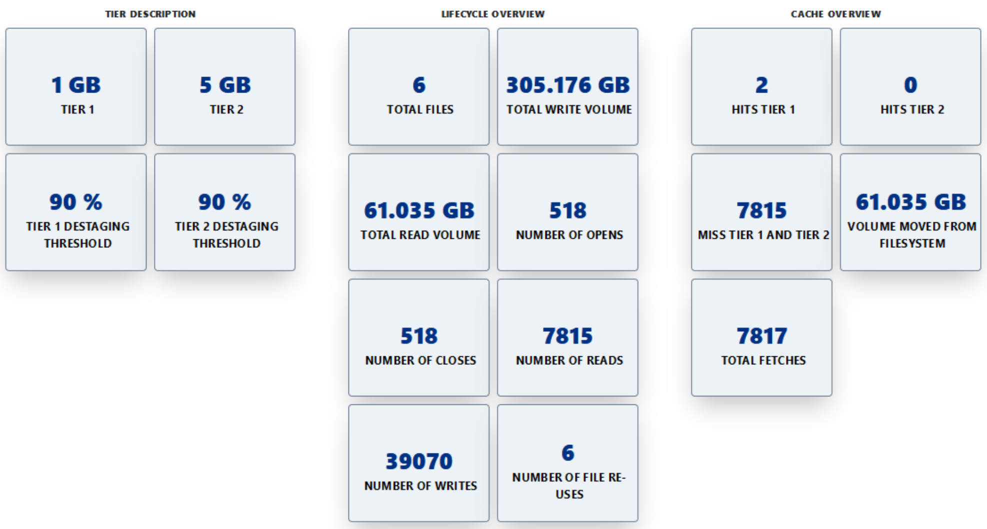
- **Cache hits from tiers**: For every tier available in the storage hierarchy, the software provides the number of

Listing 1: Example of file lifecycle declaration

```
"file_name": [
    {
    "timestamp": "xxx",
    "type": "OPEN"
    }
]
```

Listing 2: Example of file policy

```
"file_name": [
    {
    "timestamp": "xxx",
    "from": "tier_1"
    "to": "tier_2"
    }
]
```



TIER DESCRIPTION

1 GB
TIER 1

5 GB
TIER 2

90 %
TIER 1 DESTAGING THRESHOLD

90 %
TIER 2 DESTAGING THRESHOLD

LIFECYCLE OVERVIEW

6
TOTAL FILES

305.176 GB
TOTAL WRITE VOLUME

61.035 GB
TOTAL READ VOLUME

518
NUMBER OF OPENS

518
NUMBER OF CLOSES

7815
NUMBER OF READS

39070
NUMBER OF WRITES

6
NUMBER OF FILE RE-USES

CACHE OVERVIEW

2
HITS TIER 1

0
HITS TIER 2

7815
MISS TIER 1 AND TIER 2

61.035 GB
VOLUME MOVED FROM FILESYSTEM

7817
TOTAL FETCHES

Fig. 1: Example of main menu general statistics for S3DIO application.

hits. This metric allows us to assess a policy's ability to place hot files in the most suitable tier (the one delivering the best performance) and place files less likely to be accessed often in slower tiers.

- **Total fetches**: This is an additional information on total fetches to put the previous metrics in context. It corresponds to the sum of the fetches from each tier.
- **File re-use**: This metric allows us to determine the relevance of a file-level policy for a given application. For example, an application which does not re-use many files will not benefit from a file-level policy and vice-versa for applications with high file re-use.

D. Expected input format

To provide a generic API and to accommodate a wide variety of monitoring systems, we define a standard interface of files to use for our tool. Three different files are required as input for such a visualization :

- **File lifecycles**: File that contains all the operations for each file used by the application. The expected format is presented in listings 1.
- **Data movements**: File that describes data movements between the different tiers. The expected format is presented in listings 2.
- **Storage tiers metadata**: Metadata file that contains the size of the tiers and the maximum memory size allowed for the application use. Optionally, the threshold used for triggering the cache eviction can be specified.

Any file respecting this standard can be imported and visualized through the Web interface.

E. Architecture and implementation choices

The architecture of FiLiP is available in figure 2. The first step is the extraction of the raw data describing the file lifecycles, the memory movements and the storage tiers metadata. This data is then given to the file-extractor module that produces JSON files with the format described in listings 1 and 2. Once these files are created, they are given to the front-end interface that renders the profiling information for the application. This front-end interface communicates through a REST HTTP API that returns for each visualization the required data.

The Web front-end is developed using the reactive javascript framework *Vue.js* [3] and the visualization components developed using the *D3.js* library. All computations rely on a REST API, developed using the Python framework FastAPI [1].

TABLE II
FILE MANIPULATION BEHAVIOR FOR THE THREE CASE STUDIES

Application	Total files	File re-used	Number of read	Number of write	Number of open	Number of close
NEMO	56	22	1600	9904	106	106
S3DIO	6	6	6274	31365	38	38
NAMD	14	10	1346	3959	244	244

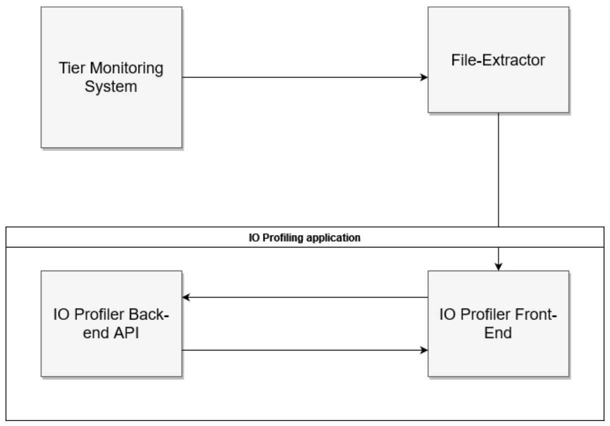


Fig. 2: FiLiP's architecture

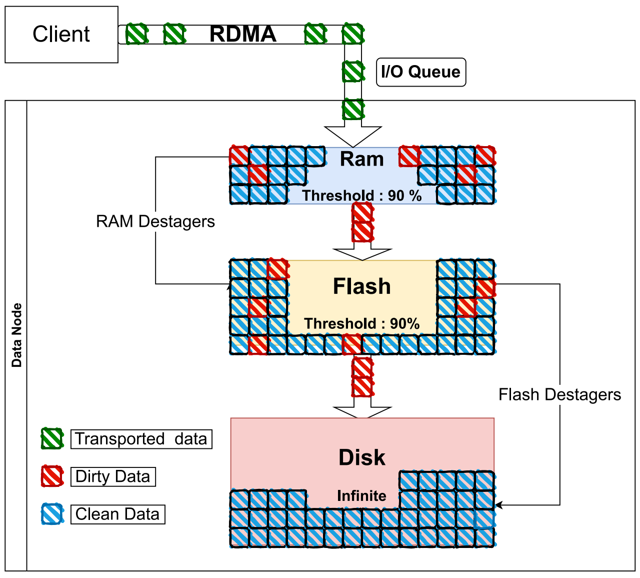


Fig. 3: Schematic representation of the burst buffer

IV. CASE STUDIES

To highlight the usefulness of FiLiP, we test it on a three levels hierarchical storage (RAM, NVMe, HDD). The RAM and NVMe are deployed as a data node, also called *burst buffer* [20] [27] [6], a fast intermediate layer located between the compute nodes and the end storage. A monitoring system is deployed on the burst buffer to capture file transfers through the storage hierarchy and observe the I/O flow between the different layers, as displayed in figure 3. When data comes to a cache level, the data has not yet been flushed to the underlying tier and is labeled as dirty. As soon as it is flushed, it is flagged as clean and can be evicted from the cache level.

To show the usefulness of FiLiP on production use-cases, we select two scientific applications and a popular I/O benchmark: NEMO [?], NAMD [23] and S3DIO [?] which implements the I/O kernel of the S3D HPC application. Because of their many parallel accesses and their high file re-use rate, these applications are relevant to show the usefulness of FiLiP and representative of the behavior of I/O intensive HPC applications. All of these applications are run on the *burt buffer* using different tiers size, to display the impact of the tier size on file movements and consequently application's performance, that can only be detected through FiLiP.

A. Experimentation scenarios and hardware

Each applications has been selected for its specific and representative behavior of sub-classes of HPC applications. Indeed, NEMO generates a large number of files and exhibits a high degree of file manipulation and re-use, S3DIO is an I/O intensive application generating a low number of files, and NAMD is very sensitive to hardware variation due to large I/O bursts. Table I summarizes the file manipulation characteristics of these three applications.

Each application is run using FiLiP according to three scenarios with a fixed combination of the available space on tiers 1 (RAM) and 2 (NVMe), and an infinite size for tier 3 (HDD) as detailed on table II. The variation in size of the different tiers shows how FiLiP can help to understand the reasons behind file movements and policy efficiency when hardware parameters or cache policy are subject to change.

TABLE II
TIER SIZE FOR EACH SCENARIO

Scenario	RAM Size	NVME Size
1	1 GiB	5GiB
2	32GiB	500 GiB
3	100 GiB	1024 GiB

Table III gathers the hit rates for the two first tiers and the miss rate when data is neither in the first tier nor in the second. We have chosen scenarios that show the difference of hit rates for the applications when we change the size of the fastest tier. So we can study the evolution of the hit rate when we present new cache movement policies in further works.

The eviction policy used in the three scenarios is *Least Recently Used* (LRU) cache management policy [16], parameterized to be triggered when a threshold of 90% of the physical capacity is reached. When this limit is reached, the dirty data is evicted from the filled cache level.

Every applications are run on a single node, with 134GiB memory, an AMD EPYC 7H12 processor with 64 cores. The data node RAM is a DDR4 and the SSD is an NVMe. The HDD storage bay relies on the Lustre filesystem.

TABLE III
Comparison of file movements statistics per ap- plication and scenarios

		RAM hit rate	Flash hit rate	Miss
Sc 1	NAMD	13.39 %	10 %	76%
	NEMO	99%	0%	1%
	S3DIO	0.1%	0%	99.9%
Sc 2	NAMD	27%	0%	73%
	NEMO	99%	0%	1%
	S3DIO	0.1%	0 %	99.9%
Sc 3	NAMD	27%	0%	73%
	NEMO	99%	0%	1%
	S3DIO	0.1%	0%	99.9%

B. NEMO

NEMO [**?**] (*Nucleus for European Modeling of the Ocean*) is a state-of-the-art modeling framework for research activities in ocean and climate sciences. It is characterized by a significant file re-use, highlighting the importance of a custom file placement policy in the hierarchical storage to keep the most accessed files in the most efficient tier.

a) Application configuration: For our experiment, we use the GYRE configuration, which simulates the seasonal cycle of a double-gyre box model, and which is often used for I/O benchmarking purpose as it is very simple to increase grid resolution and does not require any input file. In our case, the grid resolution is set to 5 and the number of MPI processes to 32 to increase the I/O activity.

b) Characterization of file lifecycle behavior: For this configuration, the application generates a total of 56 files, that are opened and closed 106 times, to realize 9904 writes, and 1600 reads. Over the 32 manipulated files, 10 are re-used which represent a ratio of 32%. Regardless of the used scenario, we observe in table III a hit rate of 99% for the first tier, as the whole dataset fits in tier 1. The misses are due to the cold accesses performed at the beginning of the application's execution, corresponding to the first access of these files on the filesystem before moving them to the highest performing tier.
The lifecycle behavior of NEMO, displayed in figure 4a, shows that the application concentrates its writes mainly on output files, such as `ocean.output` and `output.namelist.dyn`, performing several checkpoints within the application's lifetime. The configuration file `namelist_cfg` is often re-used as well, this time through read-only accesses, accessed at different moments during the application execution. Files corresponding to the checkpointing of the simulation grid (such as `GYRE_*` files) or describing the state of the mesh grid (such as `mesh_mask_*` files) are accessed more sporadically for computation purposes, and always within a short timespan.

c) Consequences on file placement policy: Despite its very high hit rate in this configuration, this application illustrates the need for a file lifecycle-based policy to manage file placement, especially if considered in a setting generating a higher volume of data. As we can see from figure 4a, results files, recognizable by the `*output*` regexp in their filenames, such as `output.namlist.dyn` and `ocean.output`, are moved frequently between the tiers, while a policy based on access frequency could have kept these files in the highest performing tiers until the end of the application execution.
On the other hand, checkpointing files (such as `GYRE_*` files) are only accessed for the duration of the checkpoint, and should be evicted directly as soon as written in the last tier to free some memory. As we can see from this example, this priority-based cache eviction policy, selected from the re-use rate, is not taken into account by the LRU policy available within the tested *burst buffer* setting and used for our experimentation. This leads to the eviction of data from the higher tier that can be re-used in the future, causing an increase of data access latency. This analysis on a file per file basis can only be done through tools like FiLiP.

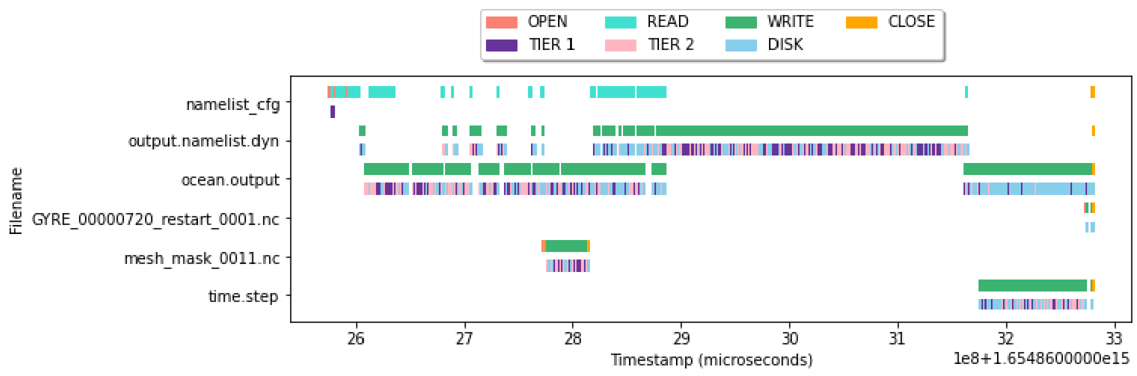
C. S3DIO

S3DIO is an I/O benchmarking application corresponding to the I/O kernel of the S3D application, a continuum scale first principles direct numerical chemical 3D-simulation code. It is an I/Os intensive application with the highest number of operations (6274 reads and 31365 writes) performed on only 6 files (against 56 for NEMO and 14 for NAMD), as can be read from table I. All its files are re-used which shows the interest of keeping them in the higher tiers for as long as possible, and will be a good case study for a further fine anticipated placement strategy of these files according to the time sequence of their re-use.

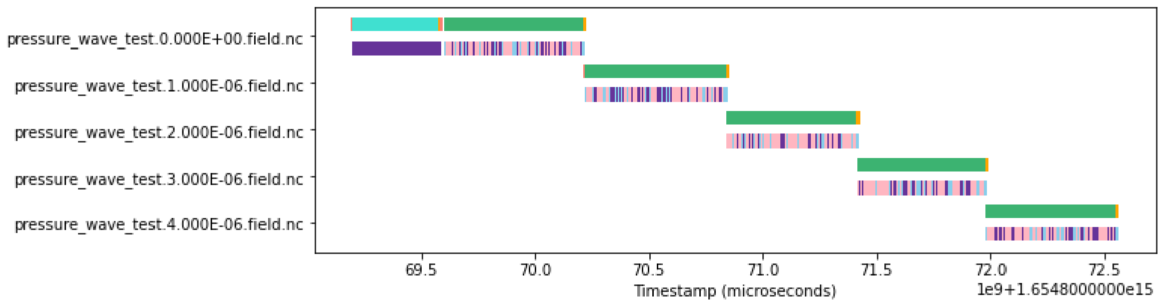
a) Application configuration: Each axis of the three dimensions were set to 800 in order to perform the computations on a large cube and increase the I/O activity of the application. For each of these dimensions, we use four MPI processes. We used a PnetCDF blocking API that allows users to first post multiple requests and later flush them altogether in order to achieve a better performance instead of a Nonblocking API. The restart parameter is set to true in order to reuse a previously written file and obtain more reads operations. The number of checkpoints, which corresponds to the number of output files as well, is set to 5 to obtain large I/Os bursts.

b) Characterization of file lifecycle behavior: The lifecycle behavior of the S3DIO application is representative of an I/O intensive application which puts each of its manipulated files under pressure during all the execution time. In the first scenario, we observe a negligible hit rate close to 0, despite high file re-use, because the accessed blocks of the files are changing all the time. We also observe that when the RAM fills up the data starts to be evicted to flash and the same phenomenon is observed from the flash to the disk. In the second scenario, the hit rate is still very low, even if the higher tiers size has have been increased due to a low blocks re-use. The third scenario leads to a slightly different behavior: the RAM (tier 1) size is large enough to contain all data, but the hit rate is still the same. As can be seen from figure 4b, we can observe that files corresponding to the grid description, characterized by their `*nc*` extension, are accessed successively in long sequences of writes, except for the first file `pressure_wave_test.0.000E+00.field.nc` corresponding to the re-use of the previously written result file.

(a) Nemo File lifecycles in the context of scenario 1



(b) S3DIO lifecycles in the context of scenario 1



(c) NAMD File lifecycles in the context of scenario 1

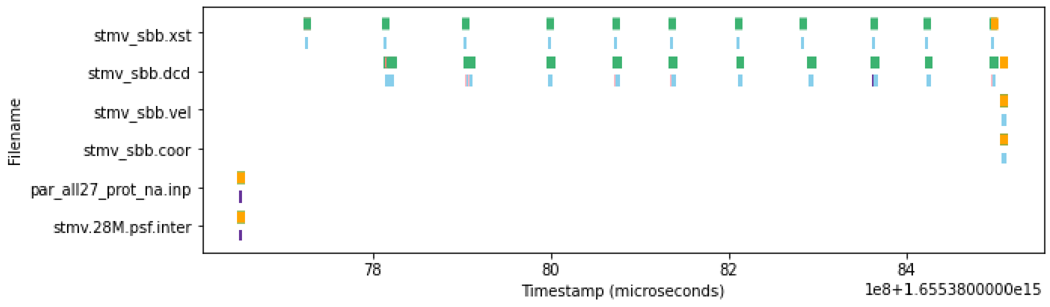


Fig. 4: Lifecycles of the tested case studies

c) Consequences on file placement policy: We can observe that the hit rate is very low because the cold access are the majority. Thus, although the data already read once are entirely in memory, the absence of a prefetch mechanism in the current implementation of the *burst buffer* causes the hit rate to remain very low. This could be improved by a lifecycle based policy that prefetches data from disk when the file is heavily re-used to predict future accesses.

D. NAMD

NAMD [23] is a parallel molecular dynamics code designed for high-performance simulation of large bio-molecular systems. It has the particularity of being very dependent on the storage hardware, due to its large I/O bursts, and is thus a good use-case for FiLiP.

a) Application configuration: For our experiment, we use the Satellite Tobacco Mosaic Virus (STMV-28M) configuration. This is a 3x3x3 replication of the original STMV dataset from the official NAMD site, containing roughly 28 million atoms. NAMD execution goes through 50 steps corresponding to the number of simulation time steps to achieve. Another parameter defines the number of steps after which a checkpoint is performed that is set to 5 to obtain ten checkpoints per run for a significant I/O activity.

b) Characterization of lifecycle behavior: This application has the highest number of file activation, with 244 open and close, and a partial but high file re-use rate, as 10 files are re-used out of the 14. The lifecycle behavior of this application provides several interesting file manipulation cases. We observe that NAMD makes I/Os on many files, each with a constant re-use rate. This results for each file in small sequences of (open-read/write-close) as we can see in figure 4c. We can observe also that the re-use rate is high due to a systematic succession of operations on each file since its

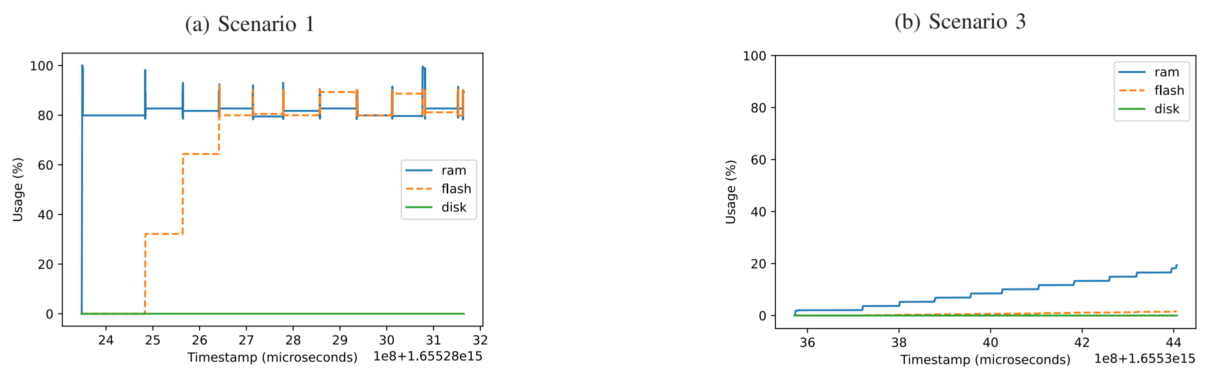


Fig. 5: Evolution of tiers fill for NAMD application

opening. NAMD handles a total of 14 files with a re-use ratio of 71%, that are opened and closed 46 times, for a total 1346 writes and 3959 reads. The application reads a total amount of 2.041 GB of data and gives different hit rates according to the different scenarios and the size of the hierarchical storage.

In the first scenario, we observe a hit rate of 13% because the RAM size is not big enough to contain all the data previously read. The 76% of miss rate corresponds to the first load of data from disk when it is read for the first time. An identical behavior is observed for the second and the third scenarios when all data fetched from disk fits in RAM and the miss rate is only due to first time cold accesses. All the data accessed a second time are already in the most performant tier.

Adding to that, we can observe that the input data, such as `stmv.28M.psf.inter` and `par_all27_prot_na.inp`, are accessed once at the start of the application and never re-used. Other files, such as checkpointing files like `stmv_sbb.xst` and `stmv_sbb.dcd` are heavely re-used.

c) Consequences on file placement policy: In the case of NAMD application, the hit rate can be increased by a data movement policy centered on the file lifecycles: a detection of the heavily re-used files would have outperformed the LRU. Similarly to S3DIO, the hit rate is low because of cold access and the absence of a prefetching mechanism. By using a predicting model that classifies input/output files we could evict this type of input files directly after their use. The checkpoint files are re-used and should be kept in RAM in priority while input ones should be evicted.

d) Impact of file placement on tiers usage: FiLiP also gives the possibility to visualize tiers filling over time. This feature helps us to understand when and why the tier eviction policy is triggered. In the case of our particular implementation of a *burst buffer*, the file movement policy fills in priority the fastest tier: tier 1, and tier2 is synchronized to the tier1 immediately such that it contains the same data.

Once they are removed from the most performant tier, they are still present in the second one and can only be evicted through a threshold based policy. As we can see from figure 5, every time the 90% threshold is reached, the data is evicted from the tier. This visualization allows us to determine how an application reaches the limits of the most efficient tiers.

In figure 5a, we can see that tiers 1 and 2 are very critical resources for the NAMD application. Indeed, when the tier usage threshold of 90% is reached, the data is evicted until another low threshold of 80% is reached. This data is evicted only when it is not dirty anymore, and an inefficient cache movement policy could stall the application while the data is awaiting eviction. In figure 5b, we present the evolution within scenario 3, where the higher tiers are large enough to contain the whole data, and therefore the eviction policy is never triggered. This confirms that an efficient file placement policy for hierarchical storage should necessarily take into account the size of the available storage, and especially for restricted resources. For NAMD application, the RAM hit rate goes from 27% in the case of a large RAM (scenario 3), as displayed in table III, to 13% in the case of smaller one (scenario 1).

V. Conclusion and further works

In this work, we have presented a file-based profiling tool called FiLiP to consider I/O from another perspective in the case of hierarchical storage, by giving the possibility to investigate file re-use properties present in HPC and scientific computing. This tool allows the understanding of how these files are used by the applications during their entire life cycle and the consequences of these re-use on the file movements through the hierarchical storage. Using FiLiP, we analyze and describe the file behavior of 3 different HPC applications: NEMO, S3DIO and NAMD, and give some interesting insights to better understand file manipulations and allow in the future a smarter file placement policies with reduced miss rates.

References

[1] Fastapi. https://fastapi.tiangolo.com/.
[2] IO Instrumentation. https://atos.net/wp-content/uploads/2018/07.
[3] Vue.js. https://vuejs.org/.
[4] A. ben Ameur, A. Araldo, and T. Chahed. Cache allocation in multi-tenant edge computing via on-line reinforcement learning. In *IEEE ICC*, 2022. DOI: 10.48550/arXiv.2201.09833. arXiv:2201.09833.
[5] 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. *ACM Transactions on Storage*, 7:1–26, 2011. DOI: 10.1145/2027066.2027068.
[6] R. F. da Silva, S. Callaghan, and E. Deelman. On the use of burst buffers for accelerating data-intensive scientific workflows. In *Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science - WORKS '17*, pages 1–9. ACM Press, 2017. DOI: 10.1145/3150994.3151000.

[7] M. Diener, E. H.M. Cruz, L. L. Pilla, F. Dupros, and P. O. A. Navaux. Characterizing communication and page usage of parallel applications for thread and data mapping. *Performance Evaluation*, pages 18–36, 2015. **DOI**: 10.1016/j.peva.2015.03.001.

[8] F. R. Duro, J. G. Blas, and J. Carretero. A hierarchical parallel storage system based on distributed memory for large scale systems. In *Proceedings of the 20th European MPI Users' Group Meeting on - EuroMPI '13*, pages 139–140. ACM Press, 2013. **DOI**: 10.1145/2488551.2488598.

[9] B. Elis, D. Yang, O. Pearce, K. Mohror, and M. Schulz. QMPI: A next generation MPI profiling interface for modern HPC platforms. *Parallel Computing Journal*, 96, 2020. **DOI**: 10.1016/j.parco.2020.102635.

[10] Y. Han, R. Wang, and J. Wu. Random caching optimization in large-scale cache-enabled internet of things networks. *IEEE Transactions on Network Science and Engineering*, 7(1):385–397, 2020. **DOI**: 0.1109/TNSE.2019.2894033.

[11] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan. *Learning Memory Access Patterns*. *Technical Report* arXiv:1803.02329, Cornell University, 2018. **DOI**: 10.48550/arXiv.1803.02329.

[12] G. Hasslinger and K. Ntougias. Evaluation of caching strategies based on access statistics of past requests. In *International Confernece on Measurement, Modelling and Evaluation of Computing Systems and Dependability and Fault tolerence*, 2014. **DOI**: 10.1007/978-3-319-05359-29.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 5th edition, 2012. ISBN: 978-0-12-383872-8.

[14] H. Herodotou and E. Kakoulli. Automating distributed tiered storage management in cluster computing. *Proceedings of the VLDB Endowment*, 13(1):43–56, 2019. **DOI**: 10.14778/3357377.3357381.

[15] S. Jiangand X. Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. *ACM SIGMETRICS Performance Evaluation*, 30(1):31–42, 2002. **DOI**: 10.1145/511399.511340.

[16] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. *Computer Journal*, 27(3):38–46, 1994. **DOI**: 10.1109/2.268884.

[17] G. Kestor, R. Gioiasa, D.J. Kerbyson, and A. Hoisie. Enabling accurate power profiling of HPC applications on exascale systems. In *proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, number 4, pages 1–8, 2013. **DOI**: 10.1145/2491661.2481429.

[18] J. S. Kim. *Parallel I/O Profiling and Optimization in HPC Systems*. PhD thesis, Pennsylvania State University, 2014.

[19] J. S. Kim, Y. Zhang, S. W. Son, M. Kandemir, W k. Liao, R. Thakur, and A. Choudhary. IOPro: a parallel I/O profiling and visualization framework for high-performance storage systems. *supercomputing*, pages 840–870, 2015. **DOI**: 10.1007/s11227-014-1329-0.

[20] 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)*, pages 1–11. IEEE, 2012. **DOI**: 10.1109/MSST.2012.6232369.

[21] N. Megiddo and D. S. Modha. ARC: A self-tuning, low over- head replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, pages 115–130, 2003. **DOI**: 10.5555/1090694.1090708.

[22] D-J Oh, Y. Moon, D. K. Ham, T. J. Ham, Y. Park, J. W. Lee, J. H. Ahn, and E. Lee. Maphea: A framework for lightweight memory hierarchy-aware profile-guided heap allocation. *ACM Transactions On Embedded Computing Systems*, 2022. **DOI**: 10.1145/3527853.

[23] J. C. Phillips, D. J. Hardy, J. D. C. Maia, J. E. Stone, J. V. Ribeiro, R. C. Bernardi, R. Buch, G. Fiorin, J. Henin, W. Jiang, R. McGreevy, M. C. R. Melo, B. K. Radak, R. D. Skeel, A. Singharoy, Y. Wang, B. Roux, A. Aksimentiev, Z. Luthey-Schulten, L. V. Kale, K. Schulten, C. Chipot, and E. Tajkhorshid. Scalable molecular dynamics on CPU and GPU architectures with NAMD. *Journal of Chemical Physics*, 2020. **DOI**: 10.1063/5.0014475.

[24] G. Quan, J. Tan, A. Eryilmaz, and N. B. shroff. A new flexible multi-flow LRU cache management paradigm for minimizing misses. In *proceedings of the ACM on Measurement and analysis of computing systems*, volume 3, pages 1–30, 2019. **DOI**: 10.1145/3341617.3326154.

[25] Md. W. Rahman, N. S. Islam, X. Lu, D. Shankar, and D. K. Panda. MR-Advisor: A comprehensive tuning, profiling, and prediction tool for mapreduce execution frameworks on hpc clusters. *Journal of Parallel and Distributed Computing*, 120:237–250, 2018. **DOI**: 10.1016/j.jpdc.2017.11.004.

[26] M. Rashti, G. Sabin, D. Vansickle, and B. Norris. WattProf: A flexible platform for fine-grained HPC power profiling. In *IEEE International Conference on Cluster Computing*, pages 698–705, 2015. **DOI**: 10.1109/CLUSTER.2015.121.

[27] M. Romanus, R. B. Ross, Robert, and M. Parashar. Challenges and Considerations for Utilizing Burst Buffers in High-Performance Computing. Technical Report arXiv:1509.05492, Cornell University, 2015. **DOI**: 10.48550/arXiv.1509.05492.

[28] W. Teng, B. Surendra, D. Bin, and T. Houjun. Univistor: Integrated hierarchical and distributed storage for hpc. *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 134–144, 2018. **DOI**: 10.1109/CLUSTER.2018.00025.

[29] K. R. Vaddina, L. Lefevre, and A. C. Orgerie. Experimental workflow for energy and temperature profiling on hpc systems. In *IEEE Symposium on Computers and Communications*, pages 1–7, 2021. **DOI**: 10.1109/ISCC53001.2021.9631413.

[30] K. Wu, Z. Guo, G. Hu, K. Tu, R. Alagappan, R. Sen, K. Park, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In *Usenix Conference on File ans Storage Technologies*, 2021.

[31] C. Zhong, M. C. Gursoy, and S. Velipasalar. A deep reinforcement learning-based framework for content caching. In *52nd Annual Conference on Information Sciences and Systems (CISS)*, pages 1–6, 2018. **DOI**: 10.1109/CISS.2018.8362276.

Adrian Khelili is a PhD Student at University of Paris Saclay and Atos R&D working on "Intelligent data placement on tiered storage. He has a mSc in computer science and is particularly interested in Data Science and its applications to complex systems.

Sophie Robert has a PhD in Computer Science on the application of black-box optimization method for the optimization of complex systems. She is a researcher at Atos R&D and her main research interest is the intelligent placement of files across hierarchical storage, and especially in the case of burst buffers.

Soraya Zertal is professor in computing science at university Paris Saclay and a member of the Architecture and Parallelism research group at Li-PaRAD lab. Her main research interests include parallel architectures and storage systems especially in HPC context, analytical modeling, simulation and adaptive/optimization strategies for data placement. She published several research articles, held a series of research grants and supervised Masters and PhDs students in the area.