Predicting Data Placement for High Performance Computing applications using File Lifecycles

Adrian Khelili
Eviden R&D Data Management
Li-PaRAD, UPSaclay-UVSQ, France
adrian.khelili@eviden.com

Sophie Robert-Hayek
LJK, Université Grenoble Alpes
Grenoble, France
sophie.robert@univ-lorraine.fr

Soraya Zertal
Li-PaRAD, UPSaclay-UVSQ
Guyancourt, France
soraya.zertal@uvsq.fr

Philippe Couvee
Eviden R&D Data Management
Echirolles, France
philippe.couvee@eviden.com

April 29, 2024

Abstract

The increasing disparity between computing speed and data access latency induces significant challenges in managing data storage systems configurations, particularly for massively parallel supercomputers. To address this issue, storage systems have evolved towards hierarchical architectures with different tiers that offer different performance, cost, and capacity based on specific underlying hardware technologies. This heterogeneous and hierarchical nature of storage comes with the need for an optimal data placement strategy. Existing strategies have initialy approached this problem from a block perspective, focusing on analyzing application I/O behavior. However, such approaches fail to capture the contextual usage of data.

Machine learning techniques, such as pattern recognition and trend prediction within time series data, enable the identification of repetitive bursts and prediction of future I/O events. These emerging approaches allow to proactively predict future I/O behavior to anticipate data placement and optimize storage utilization. The method suggested in this paper relies on the prediction of application's file manipulation using pattern matching time-series prediction. As we do not solely consider the application's I/O behavior from a block based perspective as traditional methods do, we rely on the prediction of file-usage to place files that are re-used within the near-future on the faster tier. We show that this approach allows for optimal data management, with an improvement in hit-rates.

To validate our approach, we conducted experiments using traces extracted from representative high-performance computing applications (NEMO, NAMD, LQCD), and an IO-Benchmark. The results we obtained are highly promising, demonstrating that in term of file prediction accuracy our proposed F-LRU (File-based Least Recently Used) achieves from 77% to 55% precision in most difficult scenarios to 100% for the others. It proves to be at least as effective as Least Recently Used (LRU), with an 1.06-fold increase in hit rate specifically in the context of the IO-Bench application. Furthermore, it also shows better results than Least Frequently Used (LFU), Adaptive Replacement Cache (ARC) and Low Inter-reference Recency Set (LIRS), increasing the hit rate by a factor of 1.94 compared to LFU, 1.33 against ARC for LQCD, 3.31 against LIRS, and 1.06 against LRU in the context of LQCD application. These findings highlight the potential of our approach to significantly enhance hierarchical storage performance, in HPC environments context. Thus, allocating memory resources based on file relevance optimizes storage use and facilitates tiered storage strategies.

KeyWords

Pattern recognition, Input/Output Prediction, Time Series, Machine Learning, Hierarchical storage, High Performance Computing

1 Introduction

The widening gap between computing and storage performance in modern supercomputers poses significant challenges in term of data management, particularly as applications become increasingly data-intensive. The extensive read and write operations executed by multiple nodes can strain the back-end storage, considerably increasing application waiting times for I/O operations to complete and their computation to resume. This is especially problematic for applications that routinely save their current state through checkpoints, resulting in bursts of writes. One potential approach to mitigate these I/O bottlenecks is the adoption of hierarchical storage, incorporating multiple storage levels, each with distinct technologies and physical characteristics. This hierarchy typically encompasses RAM, SSD-NVMe, SSDs, and HDD as back-end storage, as outlined in table 1. The diversity of these storage options explains the inherent trade-offs between performance and cost. While RAM delivers lower access latency, it also comes at a higher cost. In contrast, SSD-NVMe strikes a balance by offering significant performance gains at a more reasonable cost. SSDs provide a middle ground between latency and cost. HDDs, though slower, offer a cost-effective storage solution for handling large data volumes. While the primary objective of hierarchical storage is to enhance I/O execution time, it introduces its own set of challenges. Efficiently distributing data across different levels becomes a crucial task to reduce latency in data access.

Table 1: Comparison of DRAM, SSD (NVMe), SSD, and HDD Characteristics [15, 23]

Technology	**Cost ($/GB)**	**Throughput [R/W] (GB/s)**	**Latency (μs)**	**Relative Gap factor (to DRAM)**
DRAM	5.000	17/17	0.08	1 ×
SSD (NVMe)	0.200	8.0/5.0	20	250 ×
SSD	0.100	2.1/2.0	100	1250 ×
HDD	0.030	0.25/0.24	10,000	125 000 ×

Addressing this challenge can be approached through strategic management of the storage tiers: implementing intelligent data replacement strategies for instances when tiers become overly saturated and need clearance, and/or employing prefetching algorithms [4, 36] that predict the application's behavior, positioning data in the higher tiers that is most likely to be accessed.

Many algorithms address this challenge, ranging from traditional methods like LFU [35], LRU [25], and FIFO [12] to more advanced approaches such as LIRS [16] and ARC [13]. A growing trend involves exploring approaches that incorporate machine learning, and methods like Cacheus [30] and LeCar [39] showcase this evolving landscape. Most of these algorithms primarily analyze the application from a block perspective rather than a file perspective and this single and fine granularity significantly reduces their predictive capabilities due to the restricted valuable information. By adding a higher level of abstraction considering a file-based approach, we have a better understanding of the application behavior and simplify the tracking, organization, and management of data during migration between the tiers. In this work, we suggest switching from a block-only paradigm to a file-based one, by removing from the higher tiers blocks belonging to files that are most likely to be re-used. This work proposes a novel **file-based** approach for data placement, focusing on the concept of file re-use and representing files through their life cycles (*File Life Cycles*, **FLCs**). The FLC of a file captures the sequence of operations it undergoes during its lifetime, referred to as **FLC_events**.

In our context, compute nodes perform their I/O operations on a data node, thereby centralizing data and providing a comprehensive view of I/O operations. In contrast to the compute node, which lacks a complete perspective of I/Os performed by an application, especially at the file level, this centralization offers a more global view.

The main contributions of this work are:

- Introducing a new paradigm for data placement on heterogeneous and tiered storage, leveraging the file re-use property through the concept of FLC_events.
- Developing a novel file-based algorithm for time-series prediction based on pattern matching.

- Proposing two file-based data placement strategies: one that randomly selects files for removal, and another that employs time-series prediction to rank files based on their likelihood of being used.
- The application of this intelligent strategy on three real HPC applications and one benchmark proving its efficiency compared to LRU and LFU, LIRS and ARC, extending results obtained in a previous work [18].

This paper is structured as follow. Section 2 introduces works that are similar to ours and highlights the novelty of our suggestion while section 3 motivates our choice of using file-based prediction instead of block-based one. Section 4 presents notations and definitions followed by the proposed methods for time series prediction in section 5. Section 6 describes the experiments conducted to validate our proposition. Section 7 exposes and discusses the obtained results and section 8 concludes the paper giving some insight into future works.

2 Related works

Several block data management strategies have been proposed in the literature, including simple algorithms such as Random, LRU [35], LFU [25] and FIFO [12]. More advanced techniques have also been developed to strike a better balance between recency and frequency. For example, LRFU [10] offers a range of strategies between LRU and LFU, while ARC [13] divides the cache into recently and frequently accessed pages. Additionally, LIRS [16] introduces the concept of inter-reference recency to enhance efficiency. Despite the notable effectiveness of these strategies, we'll show that they are not suitable for efficiently moving large amounts of data. Unlike traditional methods, these algorithms base their data replacement decisions on the last reference, which can be inefficient when multiple blocks need to be evicted simultaneously. Our study suggests that their replacement mechanisms may not be optimal for handling significant data movements and explores the potential of machine learning-based approaches for more adaptive solutions. To address this challenge, the CAR and CART [5] algorithms have been proposed, leveraging parameter auto-tuning to automatically decide which hyperparameter to select. However, both methods fail to distinguish between data that has been recently accessed but will not be accessed again and can thus be evicted from higher tiers and data that will be accessed multiple times. CART addresses this issue by introducing a temporal filter, which enables distinguishing between short-term and long-term accesses, but this strategy still doesn't sufficiently differentiate between data accessed once and data reused multiple time. Recent advances in machine learning have led to the emergence of a new generation of algorithms based on reinforcement learning techniques. Among these, LeCar [39] employs a reward function to determine the relative importance of two strategies, LRU and LFU. This algorithm has been extended in Cacheus [30], which introduces a novel combination of strategies, namely SR-LRU (Scan-resistant LRU) and CR-LFU (Churn-resistant LFU). Furthermore, Cacheus dynamically adjusts hyperparameters using gradient-based stochastic hill climbing, enhancing its autotuning capabilities. Despite the effectiveness of these strategies, they are limited in their ability to optimize systems performance, as they do not take into account the unique characteristics of each file, as we will show in section 3.
Other data management solutions includes probability models [27] [40] [34] such as Markov Models (MM) that have the drawbacks of making the strong assumption that I/O requests are memory-less, which is not necessarily true in practice. Grammar based techniques allow the modelization of I/O behavior [11] to capture characteristics of the program's memory access patterns and proactively fetch data into the cache before it is requested by the processor. Other Machine Learning techniques such as decision trees have been used to predict I/O performance [20, 26].
When it comes to the prediction of I/O behavior using time series methods, [9] uses a time series model to estimate file system server load and [6] proposes a pattern matching approach for server-side access pattern detection for the HPC I/O stack. [37] and [8] use ARIMA (AutoRegressive Integrated Moving Average) models to capture both temporal and seasonal patterns in time series data, making it a popular choice for predicting I/O performance in various settings. ARIMA combines AutoRegressive, Integrated, and Moving Average components. The algorithm involves differencing the series to achieve stationarity, identifying optimal lag orders for auto-regression and moving average (AR and MA), and estimating coefficients through maximum likelihood estimation. The final model is then used for forecasting based on past observations and model parameters. However, the ARIMA model is complex,

computationally expensive and hard to parallelize, as highlighted in [41]. Moreover, parameter tuning can significantly affect prediction performance, making it challenging to achieve accurate and reliable results.
More generally, generic algorithms for time series prediction can be leveraged to predict I/O behavior, as they offer a variety of patterns detection techniques. We can for example cite the algorithm based on Symbolic Aggregate approXimation (SAX) [21] that converts the original real time serie T into a sequence of symbols belonging to an alphabet, and the Piecewise Aggregate Approximation (PAA) [2] which consists in dividing a time series into fixed-size segments and approximating each segment with an aggregated value, such as the mean or the median.

3 Motivation

One of the significant limitations associated with relying solely on block level accesses for data placement is the absence of contextual information regarding data usage. Unlike blocks, files at the application level typically represent logical units of information with inherent relationships and dependencies. Files can be categorized based on their access type, such as input files, checkpointing files, result files, and so on. By considering file-level usage patterns, data placement can take into account the context and semantics of the files, leveraging their inherent characteristics and access requirements. By predicting file usage at the file level, the decisions of file movements within the different storage tiers can be made much more efficiently: files which are likely to be accessed in the future can be prefetched into higher tiers, and symmetrically, files that are not likely to be accessed again can be evicted to lower tiers.
For these reasons, this paper focuses on migration strategies at the file level, where blocks are moved to a lower tier on a per-file basis guided by forecasts of incoming file usage.

Thus, we introduce two strategies :

- The FileRandom-LRU (**FR-LRU**) approach, involving naive random file selection when it comes for migration, will serve as a baseline for comparison. While it is straightforward, it underscores the need for a more comprehensive strategy that takes into account the file-level context.

- The File-LRU (**F-LRU**) approach, driven by predictive modeling based on historical intra-file access patterns, aligns with this motivation. By shifting the focus to files and using past usage trends, this approach aims to prioritize files based on their likelihood of future access.

These predictions enable us to establish a priority ranking for the files. The file predicted to exhibit the least activity or deemed less important based on the observed pattern is then selected for migration to a lower tier with higher priority. Traditional time series modeling approaches like ARIMA, AR, and MA are often time-consuming and require complex parameter tuning. The objective here is to provide a time efficient algorithm that competes with classical time serie prediction algorithms in terms of prediction accuracy without inducing a tuning phase. Both ARIMA [42] and KNN regression [24] from which our algorithm is inspired have their advantages and drawbacks as studied by Smith et al. [3] and cannot be used straight forward. ARIMA, as a parametric model, stands out for its effective capture of dependencies with previous observations. However, its primary drawback lies in the continuous retraining required during execution—a computationally expensive process. This need arises due to the dynamic evolution of the time series throughout the application, introducing substantial computational complexity. In contrast, KNN model emerges as an attractive alternative by eliminating the necessity for continuous learning phases. This characteristic represents a significant advantage, particularly when the time series undergoes variations during application execution. However, it's crucial to highlight that in KNN regression, the algorithm predicts the target value of a new data point by considering the K-nearest data points in the feature space. The choice of distance metric plays a crucial role in determining "closeness" or similarity between data points. This, in turn, directly influences the predictions made by the KNN regression model. The distance metric measures the dissimilarity between two data points and is a critical parameter. Different distance metrics capture different aspects of similarity, and the choice depends on the characteristics of the data and the problem at hand. In the upcoming section 5, we will discuss two specific distance metrics, namely Dynamic Time Warping (DTW) and Euclidean distance. We will examine their respective effectiveness

and computational complexity within the context of KNN (K-Nearest Neighbors) regression for file lifecycles prediction.

4 Definitions and notations

In the upcoming sections, we will use the following terms and notations:

- **Time Serie:** A time serie $\mathbf{T} = (\mathbf{t_1}, \mathbf{t_2}, ..., \mathbf{t_n})$ of length n is a sequence of n real-valued observations, where each t_i represents a value observed at a specific time point.
- **Subsequence:** Let T be a time serie of length $\mathbf{n}$. A subsequence $\mathbf{S_{i,l}}$ is a contigous time serie of length $\mathbf{l}$ with $\mathbf{1 \leq i \leq n}$ and $\mathbf{1 \leq i + l \leq n}$. $\mathbf{i}$: $\mathbf{S_{i,l}} = (\mathbf{t_i}, \mathbf{t_{i+1}}, ..., \mathbf{t_{i+l-1}})$.
- **FLC_Events**: FLC_Events are all POSIX operations such as OPEN, READ, WRITE, and CLOSE made on the same file.
- **File Lifecycles (FLC)** : a temporal series of FLC_events $(o_t)_{1 \leq t \leq T}$ performed on a given file, with o_t an operation in the POSIX set {READ, WRITE, OPEN, CLOSE}.

5 Predictive data management strategy

The main objective of our approach is to anticipate the appropriate data placement on the tiers by establishing a priority files ranking based on their likelihood of being reused or not in the near future. By accurately predicting file usage patterns, a priority ranking of files can be established at runtime. This ranking guides the decision-making process that determine which files to evict first from the higher tiers of the hierarchical storage.

5.1 Time series pre-processing

We model files as time series, representing the File Lifecycles (FLC) . As a pre-precessing phase, we

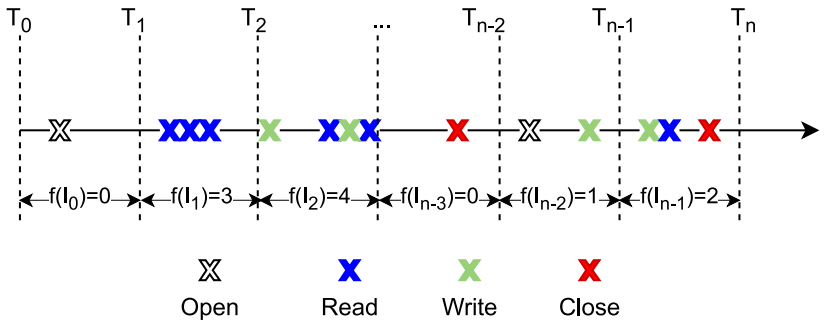


Figure 1: FLC Time serie representation

perform a binning on FLC's that involves dividing the time series into intervals $\{T_1, T2, .., T_n\}$. This fixed-size interval constitutes an hyperparameter denoted as w. This binning divides the time series into a set of bins denoted $I = \{I_1, I_2, ..., I_n\}$, where I_i represents the ith interval of the time serie. The total number of bins, n, is determined by the ratio between the total duration of the application and the bin size. Function f counts the number of requests within interval i and $f(I_i)$ represents the file access frequency at each interval i as represented in figure 1, calculated for READ and/or for WRITE operations. The "Update Time Series" algorithm operates online to maintain time series of file accesses. When a new file access request is received, the system extracts the timestamp of the request to determine the exact moment of the access. The check for the request type allows targeting specific operations such as file reads or writes. If the condition is met, the algorithm proceeds to update the global time series online, reflecting the frequency of accesses to the file system over time intervals defined by the window w. The exact time window of the request is obtained by aligning the request timestamp with the current window using the expression req_ts - (req_ts % w). Simultaneously, the

algorithm initializes or updates a file-specific time series associated with the request. It takes care to track time windows where no activity has been recorded by setting their value to 0, ensuring continuous and dynamic management of file accesses. This online approach allows real-time adaptation to changing access patterns, providing valuable flexibility in understanding and managing file system activities.

Algorithm 1 Update Time Series

```
procedure UPDATE_TIME_SERIE(files, request, req_types, w)
    Parameters:
        files: Time serie describing the FLC of each file.
        request: structure containing all the informations concerning the request.
        req_ts: The type of requests we want to filter on.
        w: Window size.
    req_ts ← request.get_timestamp()
    if request.type ∈ req_types then
        if not last_ckpt then                          ▷ Case where its the first request
            last_ckpt ← req_ts − (req_ts % w)          ▷ We define a new interval
            init_step(files, last_ckpt)                ▷ Init the FLC_Events counter to 0 for each FLC
        if req_ts ≥ last_ckpt + window then            ▷ Case where we're not anymore in the same interval
            last ← last_ckpt                           ▷ We store the last interval
            last_ckpt ← req_ts − (req_ts % w)          ▷ We define a new interval
            init_missing_steps(files, last)            ▷ Init FLC_Events counter to 0 for each missed FLC interval
        if request.filename ∉ files then               ▷ The file is seen for the first time
            files[request.filename] ← new instance of FileSerie(request.filename, w)
            files[request.filename].init_ts( last_ckpt)
        files[request.filename].update(last_ckpt)      ▷ We increment FLC_Events counter on given file
```

5.2 Distance metrics

In the case of KNN regression, a distance is a measure of similarity between two time series subsequences. We considered in this work to include both the Euclidean distance (ED) and Dynamic Time Warping (DTW) due to their unique abilities. ED, known for its computational efficiency, provides a straightforward measure of similarity [32]. In contrast, DTW accommodates variations and distortions, offering a more nuanced representation of pattern similarities and differences [17]. It's important to note that DTW can be computationally expensive due to its dynamic programming nature. By incorporating both distances, we aim to conduct a comprehensive analysis, comparing their effectiveness and considering their relative computational costs. This approach allows us to discern scenarios where computational efficiency is paramount, favoring ED, and those where DTW's unique ability to handle distortions proves essential in capturing pattern differences.

The Euclidean distance between two time series X and Y can be computed using the following formula:

$$\mathrm{ED}(X, Y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Dynamic Time Warping measures the similarity between two temporal sequences, even when they have different lengths or exhibit temporal distortions, as it searches for the optimal alignment between the two sequences by warping and stretching them in the time dimension. A distance measure is computed between the two sequences by creating a matrix that represents the accumulated cost of aligning each pair of elements in the sequences. The elements of this matrix correspond to the partial alignments of the sequences at different time points.
The formula for computing the DTW distance between two sequences X and Y of lengths n and m respectively is as follows:

$$\mathrm{DTW}(X,Y) = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m} ED(x_i, y_j)}$$

where $ED(x_i, y_j)$ is the local distance measure between elements x_i and y_j in the sequences X and Y respectively. We use here the Euclidean distance.
The DTW algorithm then finds the optimal alignment by iteratively computing the accumulated cost matrix and backtracking through it to determine the optimal path. The resulting optimal path represents the alignment between the sequences.

5.3 Suggested data placement algorithm

When we apply it to FLC prediction, this pattern matching algorithm leverages the assumption that by identifying the most similar observed FLC_events patterns, we can predict the future behavior of the application at the file level, an assumption validated by works on the periodicity of HPC applications [33] [14]. The FLC timeseries is then browsed with a sliding time window that moves through the timeseries and examines subsequences, as displayed in figure 2. This window should not be confused with the binning window w mentioned earlier. It corresponds to the pattern containing h windows w. The distance calculation function is a callback parameter *dist* to test both the DTW (Dynamic Time Warping) and ED (Euclidean Distance) calculation algorithms. Once the best matching pattern is found, it serves as the basis for prediction as described in algorithm 2. We can thus predict how the file will behave in the future based on its historical observed patterns and anticipate data placement according to the predicted behavior.

Algorithm 2 Pattern matching algorithm with time series prediction

Ensure: *prediction*: A prediction for the h following values.
function PatternMatchingPrediction(*time_series*, h, *dist*)
  Parameters:
    time_series: Time serie describing the FLC.
    h: Prediction horizon.
    block_size: Custom distance metric.
  current_pattern ← get_curr_pattern(*time_series*, h) ▷ Get the current pattern of size$h$
  closest_pattern ← None
  min_distance ← ∞
  for $i \leftarrow 0$ to len(*time_serie*) $- h - 1$ **do**
    pattern ← *time_serie*$[i : i + h]$ ▷ Extract the pattern
    distance ← dist(*current_pattern*, *pattern*) ▷ Calculate the distance
    if *distance* < *min_distance* **then**
      min_distance ← *distance*
      closest_pattern ← *pattern*
  prediction ← predict_from_pattern(*closest_pattern*) ▷ Predict from closest pattern
  return *prediction* ▷ Return the prediction

This algorithm can be conceptualized as illustrated in Figure 2. In the visualization, the blue curve represents a time series describing a file. The sliding window, denoted by the interval in red dashed lines, traverses the time series to identify the most similar pattern within the pattern delimited by black dashed lines. The duration spent between two observations represents w. This process facilitates the prediction of the h upcoming values, depicted by the green segment in the time series.

Once this process is executed for each of the files, the file with the least likelihood of being reused selected, it is also necessary to choose the specific blocks to be evicted within that file. The blocks will be evicted in the LRU (Least Recently Used) order, meaning that the least recently accessed blocks within the file will be evicted first. Our aim is just to free enough room in the tier to achieve our anticipation and load its corresponding data. As the prior file for data placement is also in use (even with the lowest reuse score), we adopt a conservative strategy and we evict only blocks to free enough space for potential data to be accessed in the future according to our prediction.

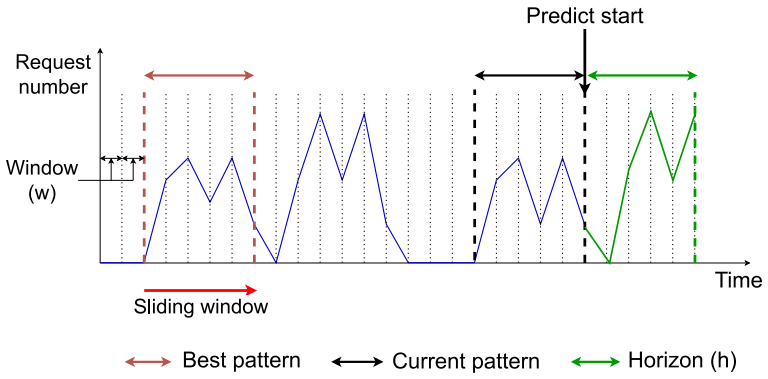


Figure 2: Schematic representation of file prediction process

5.4 Complexity of the algorithm

Let m be the number of manipulated files, p the number of patterns, FLC the file lifecycle time serie, and n = $|FLC|$, where $|FLC|$ denotes the cardinality of FLC in terms of number of windows w. The complexity of this algorithm is $O(n \times m \times p)$ when the selected distance metric is euclidean and $O(n^2 \times m \times p)$ in the case of DTW metric. There is an optimized version of the DTW algorithm that uses a subsampling strategy to reduce the time complexity to $O(n \times \log(n) \times m \times p)$ at the cost of a slight loss in precision due to the resolution reduction. Despite that faster computation time, we adopted the original DTW distance rather than fast DTW for the sake of precision. Additionally, our solution is highly parallelizable since the files behaviors are independently predicted and the comparison between patterns are independently computed. This means that a subset of opened files can be considered by our prediction in parallel (subject to the number of cores, parallel execution strategy, etc), which will be an important feature when porting our work to production. Note that by fine-tuning the granularity of comparison through specific step intervals, we can precisely prune the search space, eliminating the need to compare some patterns. This approach enables us to strike a judicious balance between prediction accuracy and computational efficiency, which is not the case with ARIMA.

6 Validation

We evaluated our data management proposal to a Burst Buffer I/O-accelerator with a heterogeneous hierarchical storage that will be detailed further in this section. We organize the validation of our prediction strategy based on pattern matching coupled with a file-based management strategy of the tiered storage in two steps: we first validate the quality of the file re-use prediction ranking by running the prediction algorithm every 10 bins, and evaluate the prediction once the actual data, which will act as a ground truth, is available. In the second phase, we evaluate and compare the performance of our migration solution with standard strategies. To measure the effectiveness of our strategy, we utilize representative metrics such as the hit ratio. By comparing the hit ratio of our method to that of standard ones, we can assess its efficiency and effectiveness.

6.1 Selected applications

Four applications have been selected to evaluate the performance of our algorithm: three HPC applications and a synthetic I/O benchmark. Table 2 presents a comprehensive summary of the I/O behaviors related to file manipulation for each application. We will confront our strategy to a variety of real-world and benchmark scenarios by selecting applications with diverse characteristics. This approach is designed to enhance the relevance of our study, providing a thorough evaluation of our strategy across different contexts and application profiles.

6.1.1 NAMD

NAMD [29] 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 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. In this configuration, a total of 2 GB of data is read and 5 GB of data is written, resulting in a total of 10226 read operations and 1079 write operations. These operations involve a total of 14 files, 10 of which are heavily re-used.

6.1.2 NEMO

NEMO [31] (*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 strategy in the hierarchical storage to keep the most accessed files in the most efficient level. Additionally, we can see that NEMO constantly manipulates a certain number of files, whether in read or write mode, which offers rich patterns for both READ and WRITE operations. It presents a higher activity in reading at the beginning of the application and a higher activity in writing at the end of the application, which is explained by the reading of input files at the beginning and the writing of output files at the end of the application.
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. In this configuration, a total of 65 GB of data is read and 50 GB of data is written, resulting in a total of 17,816 read operations and 17,816 write operations. These operations involve a total of 184 files, among which 21 are heavily reused.

6.1.3 LQCD

Lattice QCD is a well-established non-perturbative approach for solving the quantum chromodynamics (QCD) theory of quarks and gluons. It is a lattice gauge theory formulated on a grid or lattice of points in space and time. When the size of the lattice is taken infinitely large and its sites infinitesimally close to each other, the continuum QCD is recovered [7].
In this lattice quantum chromodynamics (LQCD) simulation, the parameters are set as follows: the quark mass (qmass) which represents the mass of the quark is set to 0.02, and the gauge coupling strength (beta) which represents the strength of the interactions between quarks and gluons in the lattice simulation is set to 0.2. The simulation is performed with a spatial extent of 16 lattice units and a temporal extent of 36 lattice units. The simulation is divided into a total of 8 tasks in the temporal direction, 4 tasks in the spatial direction, 4 tasks in the y-direction, and 4 tasks in the x-direction, resulting in a total of 512 tasks.
In this configuration, a total of 27 GB of data is read and 40 GB of data is written, resulting in a total of 1673538 read operations and 3335740 write operations. These operations involve a total of 284 files, among which 21 are heavily reused.

6.1.4 Benchmarking application

We deliberately designed this synthetic application using a generic I/O benchmarking tool in such a way that the read phases gradually lengthen. Except for the initial and final stages of the application, the level of activity in writing is relatively low. In this configuration, a total of 42 GB of data is read, while only 1 GB of data is written, resulting in a total of 36,137 read operations and 1,240 write operations. These operations involve a total of 9 files, all of which are heavily reused.

The synthetic application we've developed complements our study, which already includes real-world applications. It enables us to control and emphasize specific behaviors that may be challenging to isolate in complex real-world settings. This enhances our analysis especially in cases where infrequent but noteworthy phenomena occur. We've isolated and designed this application specifically chose this approach to demonstrate the differences in patterns as they unfold. Our first objective is to compare the effectiveness of Dynamic Time Warping (DTW) and Euclidean Distance (ED) as distance metrics

Table 2: I/O behavior of the four applications

Application	Total files	File re-used	Number of read	Number of write	READ volume (Gb)	WRITE volume (Gb)
NEMO	184	21	17816	12867	65	50
NAMD	14	10	1226	1079	2	5
LQCD	284	121	1673538	3335740	27	40
BENCH	9	9	36137	1240	42	1

in this particular context. Furthermore, we aim to observe a profile characterized by significant file reuse. By integrating both synthetic and real-world elements, we establish a solid foundation for evaluating distance metrics and behavior models within the broader context of our study.

6.2 Hardware and software

To capture their I/O behavior, each application is run on a single compute node, with 134GiB memory, an AMD EPYC 7H12 processor with 64 cores.

Our selected particular use-case of hierarchical storage is a burst buffer [22, 38], a fast intermediate layer located between compute nodes and the slower backend storage, as displayed in figure 3. Their purpose is to allocate a temporary cache positioned between the computing processes and the permanent storage system. Burst of I/O can be redirected to this high bandwidth cache to avoid I/O bottlenecks during intensive I/O phases, such as checkpointing phases. The burst buffer absorbs burst of I/O and asynchronously performs the I/O on the permanent storage backend to reduce the job's stalling time and accelerate its execution. In our particular implementation, two layers of intermediate faster storage are available: RAM and NVME.

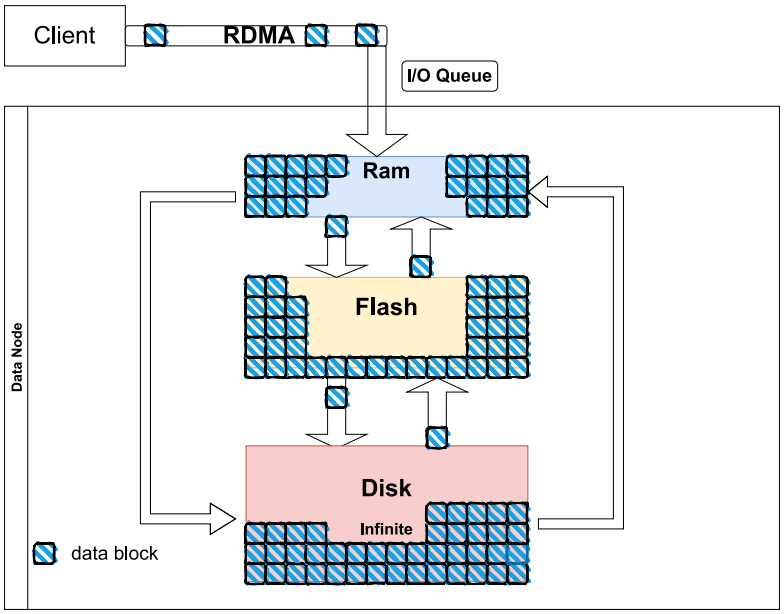


Figure 3: Atos data node internal architecture [1]

In Figure 3, we depict the blue blocks as I/O requests arriving from the compute nodes. Also note that we assume the quantity of HDD storage to be infinite to simplify the task, as we do not have any capacity constraints on this device.

6.3 Prediction quality evaluation

In this validation phase, we conduct an in-depth comparison of the file ranking quality achieved by our algorithm in contrast to that produced by the ARIMA model. To ensure a thorough evaluation of these two approaches, we have opted to employ relevant performance metrics. Given the nature of file ranking

in this context, we have carefully defined specific metrics tailored to our ranking algorithm. Our choice of metrics is guided by the need to capture the nuances of file ranking performance. We recognize that traditional metrics may not fully reflect the intricacies of our algorithm's output. Therefore, we have formulated metrics that align with the objectives and characteristics of our ranking method. This tailored approach allows us to provide a nuanced and precise evaluation, ensuring that the selected metrics appropriately measure the success and effectiveness of our file ranking algorithm in comparison to the ARIMA model.

We calculate standard multi-class supervised learning metrics in Machine Learning: accuracy, precision, and MCC for Matthews Correlation Coefficient [6]. We subdivide the order of file prediction into four quartiles, and treat each of these as a specific category. We define for each quartile:

- True Positives (TP) : number of correctly predicted instances for a specific quartile
- True Negatives (TN): number of correctly predicted instances for all other classes excluding the specific quartile
- False Positives (FP) : number of incorrectly predicted instances as the specific class
- False Negatives (FN) : number of instances belonging to the specific class but incorrectly predicted as other classes.

The different metrics are calculated and averaged for quartile:

Accuracy, Precision, and the Matthews Correlation Coefficient (MCC) serve as crucial metrics to evaluate the performance of a classification model.

Accuracy serves as a holistic metric, offering an overall measure of correctness in predictive models. It quantifies the average proportion of correct predictions across all quartiles. The calculation involves finding the ratio of the sum of true positives (TP_i) and true negatives (TN_i) for each quartile over the total number of predictions, whether they are true or false. We then divide this sum by the total number of quartiles. In essence, accuracy provides a comprehensive assessment of the model's ability to make correct predictions.

$$\text{Accuracy} = \frac{1}{4} \times \sum_{i=1}^{4} \frac{TP_i + TN_i}{TP_i + FP_i + FN_i + TN_i} \tag{1}$$

Precision specifically evaluates the accuracy of correct positive predictions made by a model. It's calculated for each class by dividing the true positives by the sum of true positives (TP_i) and false positives (TP_i). The average precision across all classes provides a consolidated measure of the model's capacity in making correct positive predictions.

$$\text{Precision} = \frac{1}{4} \times \sum_{i=1}^{4} \frac{TP_i}{TP_i + FP_i}$$

The **Matthews Correlation Coefficient**, expressed in the given formula, combines information on true positives, true negatives, false positives, and false negatives, providing a comprehensive measure that considers the balance between sensitivity and specificity. With a scale ranging from -1 for a completely incorrect classification to 1 for a perfect classification, MCC offers a nuanced assessment of the model's performance, capturing both correct and incorrect predictions across different classes.

$$\text{MCC} = \frac{1}{4} \times \sum_{i=1}^{4} \frac{TP_i \cdot TN_i - FP_i \cdot FN_i}{\sqrt{(TP_i + FP_i)(TP_i + FN_i)(TN_i + FP_i)(TN_i + FN_i)}} \tag{2}$$

For example, an Accuracy of 100% means that every file was predicted in its right quartile. We check the quality of our prediction via these metrics each $w \times 10ns$, then average the results over the whole application execution.

6.3.1 Algorithm hyperparameters

Our prediction also requires two hyperparameters. One of these hyperparameters is the window size (w), which is used to determine the number of I/O requests to consider when creating the time serie. The window size determines the granularity of the time serie, i.e., the frequency at which FLC_events are aggregated to form the observation points where each observation $f(I_n)$ refers to the number of FLC_events made within that specific interval I_n. The length of the subsequence that describes the pattern is also a hyperparameter. The number of predicted values is denoted by the horizon (h). We select for our experimentation campaign a value of 40 as a trade-off between the quality of the prediction (the closer the prediction the easier it is to predict accurately) and the number of predicted points (the more information we have, the more efficiently we can predict data placement). The window size (w) corresponds to the width of the pooling interval to transform: in production, it should be carefully chosen based on the shape of the bursts of the application, as it impacts the shape of the timeseries and subsequently affects the quality of predictions. For this work, we select (w) based on the execution time of the strategy, to have 1000 data point in total within the time series. The value is then rounded to the nearest power of 10. A summary of tested hyperparameters is available in table 3.

Table 3: Tested hyperparameters per application

Application	h	w
LQCD	40	10 000 000
NEMO	40	1 000 000
NAMD	40	100 000
Synthetic App	40	100 000

6.4 Hit ratio evaluation

In the case of our datanode, once the maximum size of a tier is reached, a data placement process is initiated. This operation involves freeing up space by removing data from the respective tier until the capacity reaches 80%. In other words, when the tier's capacity is fully utilized, we initiate the removal of data to maintain a 20% margin of available capacity. This approach ensures dynamic storage space management, facilitating efficient resource utilization while preventing situations of saturation.In our evaluation of data placement management quality, we expand our focus to include the assessment of hit rates on individual tiers, taking into consideration the variability in access latencies. To establish a baseline for optimal cache performance, we start with an infinite cache scenario, calculating the perfect theoretical hit ratio. This calculation, excluding cold access-related misses, provides insights into the upper limit of cache efficiency. Subsequently, we conduct experiments across three scenarios with different cache sizes, as outlined in Table 4, with a specific emphasis on evaluating the practical efficiency of our data placement management strategy on every tier. The hit ratio calculations for each tier in these scenarios offer a nuanced understanding of our approach's performance under varying cache size constraints. These results are then compared to established data management algorithms, such as LRU, LFU, LIRS, and ARC, to assess the effectiveness of our approach in multi-tier cache environments.

The experiments involve three distinct configurations with varying hierarchical storage characteristics: the small configuration featuring 1 GB RAM and 5 GB NVMe, the intermediate configuration with 10 GB RAM and 50 GB NVMe, and the large configuration equipped with 100 GB RAM and 500 GB NVMe.

Table 4: Selected hierarchical storage characteristics for our experiments

Configuration	RAM (Level 1) Size	NVME (Level 2) Size
Small	1 GB	5 GB
Intermediate	10 GB	50 GB
Large	100 GB	500 GB

6.5 Implementation

The extraction of the File-Level Characteristics (FLC) for each application is performed through the FiLiP software, as introduced by Khelili et al. in [19]. This software enables the extraction, on a file-per-file basis, of crucial details such as the operation type, starting offset, volume in bytes, and timestamp, as illustrated in Listing 1. Consequently, all operations carried out by the application are extracted for each file. However, the switch in paradigm from block-oriented to file-oriented requires an implementation that doesn't introduce additional complexity. Therefore, we propose managing access volumes at the block level taking into account the file dimension and introducing a data structure tailored to address our specific problem.

Listing 1: Example of FileLifecycle

```
        "file_1": [
            {
            "timestamp": 1657179154904224,
            "type": "WRITE",
            "offset": 0,
            "size": 13000,
            }, ...],
        "file_2": [
            {
            "timestamp": 1657179154904224,
            "type": "READ",
            "offset": 13000,
            "size": 500,
            }, ...],
        ]
```

6.5.1 I/O volumes management

In high-performance computing (HPC) applications, the sizes of I/O operations often vary significantly. Recognizing this diversity in data volumes, we have implemented a systematic approach to address the challenge. In response to the varying sizes of I/O requests, our solution involves segmenting these operations into uniform-sized blocks. This strategic segmentation allows for the standardized processing of I/O requests, offering a consistent framework that accommodates the inherent differences in data magnitudes. By dividing each I/O request into blocks of constant size (4Kb), we mitigate the challenges posed by disparate data volumes. This approach not only streamlines data operations but also optimizes storage resource utilization by providing a predictable structure for each block. To consistently access these blocks, it is necessary to define a hashing format that uniquely identifies each block for each file, adopting the form "filename[start-end]". In order to create these blocks hash, we employ the following algorithm 3.

For this, we calculate the start and end for each of the blocks based on the offset and size parameters. Each file is virtually divided into blocks of *block_size*. Thus, an I/O operation can involve one or more blocks of the file. Two cases are possible: either the request accesses only a single block of size *block_size*, and a single hash string is returned, or it accesses multiple blocks, and the function takes care of returning, for each of the blocks, the corresponding hash key via the array of strings *hashes*. Each of the contained keys uniquely identifies a specific block of a particular file. This file block division will facilitate updating the data structure described in subsection 6.5.2. Indeed, This systematic approach to hash label generation facilitates rapid and targeted access to different segments of the file used to update the underlying LRU structure. It is often necessary to move a block to bring it to the front of the queue, etc.

Let's take the example of the first I/O made on listing 1 where the "file1" is accessed by a request with an offset ranging from 0 to 13000. In this case, the hash label generation function, designed for blocks of 4 KB, would generate hash labels for the distinct blocks within this range, resulting in identifiers returned as a list such that : ["file1[0, 4096]", "file1[4096, 8192]", "file1[8192, 12288]", and

Algorithm 3 Hash Label Generation

Ensure: *hashes*: List of hash labels

function GENERATEHASHLABELS($offset, size, block\_size, filename$)
  $hashes \leftarrow []$
  Parameters:
    $offset$: Starting position of the I/O operation.
    $size$: Number of elements in the I/O operation.
    $block\_size$: Size of the blocks used for segmentation.
    $filename$: Name of the file on which the I/O operation is performed.
  $start\_block \leftarrow \text{Integer}(\frac{offset}{block\_size})$
  $remove\_block \leftarrow 0$
  if $((offset + size)\%block\_size) == 0$ **then**
    $remove\_block \leftarrow 1$
  $end\_block \leftarrow \text{Integer}(\frac{(offset+size)}{block\_size}) - remove\_block$
  if $start\_block = end\_block$ **then**
    $start \leftarrow start\_block \times block\_size$
    $end \leftarrow (start\_block + 1) \times block\_size$
    $block\_label \leftarrow \text{String}([filename[start - end]])$
    $hashes.append(block\_label)$
  else
    $hashes \leftarrow []$
    for $iter$ from $start\_block$ to end_block **do**
      $range\_start \leftarrow iter \times block\_size$
      $range\_end \leftarrow (iter + 1) \times block\_size$
      $block\_label \leftarrow \text{String}([filename[range\_start - range\_end]])$
      $hashes.append(block\_label)$
  return $hashes$
$=0$

"file1[12288, 16384"]. Each of these hash labels uniquely identifies a specific 4 KB block within the file. In the case of the request made on file2, it returns ["file2[12288,16384"]].

6.5.2 Data structure

For quick simulation and comparison of our data movement strategy to the state of the art, we have developed a simulator that processes requests as they arrive and simulates the execution of the data management strategy. We implemented LRU and LFU, LIRS and ARC in their most optimal version, O(1), for all operations to allow fast experimentations. Both FR-LRU and F-LRU strategies also required the creation of a specific data structure to achieve O(1) execution for all operations in order to avoid overhead due to the unit switch from the block to the file.

Our data structure figure 4 is built upon the foundation of the LRU data structure (in black), incorporating a combination of a linked list and a hash map. This design allows an efficient insertion, deletion, and update operations with optimal time complexity. The linked list component is responsible for organizing the blocks based on their recent usage, notably, facilitating the rapid update of blocks. It maintains the order of blocks, placing the most recently accessed block at the front of the list and the least recently accessed block at the back. The hashmap part enables access to blocks through their hash keys generated by the algorithm previously explained in 3 and represented by key_1, key_2,..., key_n in the data structure. This configuration ensures that migrating the least recently used blocks to a lower storage tier can be accomplished in constant time. This is achieved by seamlessly removing the block from the tail of the linked list in the first tier and inserting it at the head of the linked list in the second tier. To further enhance our file-level strategy, we recognized that updating and deleting blocks belonging to the same file might result in a time complexity of O(n), where n represents the number of blocks. To address this concern, we introduced an additional level (green links) of the linked list that connects blocks belonging to the same file. This allows us to efficiently update or delete blocks associated with a specific file by traversing only the linked list related to that file, resulting in improved performance. The simulator takes as input the files lifecycles of an application, and returns

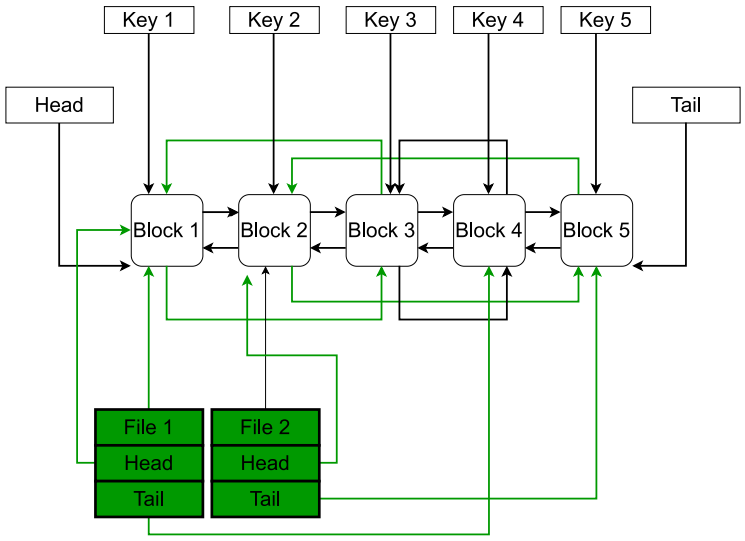


Figure 4: File-based data management structure

the number of hits for the different tested strategies.
The capacity of every tier of the hierarchical storage configuration can be adjusted, as well as the block size.

6.6 Baseline algorithm FR-LRU

In order to contextualize the performance of our algorithm File-LRU (F-LRU), we also proposed a comparative baseline algorithm FileRandom-LRU (FR-LRU). This baseline involves a random prioritization strategy for files, where each file is assigned a random priority. The algorithm then proceeds to remove blocks based on the Least Recently Used (LRU) order within the files with randomly assigned priorities. This approach serves as a reference point for evaluating the efficacy of our primary algorithm. By comparing the results obtained from our algorithm against this random prioritization baseline, we aim to showcase the advantages and improvements brought about by our proposed solution. The random prioritization strategy offers a contrasting perspective, allowing us to discern the specific contributions and efficiencies achieved by our algorithm in the context of file and block movements. The principal difference with File-LRU (F-LRU), lies in the prioritization strategy. While FR-LRU adopts a random prioritization approach for files, assigning each file a random priority, F-LRU relies on a more sophisticated approach. Our strategy uses a prediction based on the pattern matching algorithm we developed, providing a smarter prioritization of files based on past usage trends. By comparing the results obtained from our algorithm against those of the baseline, we aim to highlight the substantial advantages and improvements brought about by our proposed solution.

7 Results and discussion

Figures from 5 to 8 represent the FLC's of the three studied scientific applications and the benchmarking one. Each color in these figures represents a different FLC belonging to a specific file. We can thus see that certain files exhibit specific behaviors. For each application we observe its prediction quality using accuracy, precision, and MCC metrics. The distinct characteristics of files in figure 6b make it easily recognizable that the purple file serves as an input file. Its prominent activity at the beginning of the application, followed by a lack of ongoing activity, strongly suggests its role in providing initial input data. Similarly, when examining files like the brown one in figure 5b, a consistent level of activity throughout the entire application execution indicates its nature as a checkpoint file, with recurring accesses. Furthermore, there is a file depicted in yellow in figure 5b where activity is concentrated solely at the end of the application. This pattern suggests that this particular file serves as an output file, capturing the results or final data generated during the execution. It's worth noting that these file behavior patterns observed for NAMD are not unique to this application; rather, they are indicative

of broader categories of file behaviors that are consistent across the various applications under study.

7.1 NAMD results

(a) NAMD READ Behavior

(b) NAMD WRITE Behavior

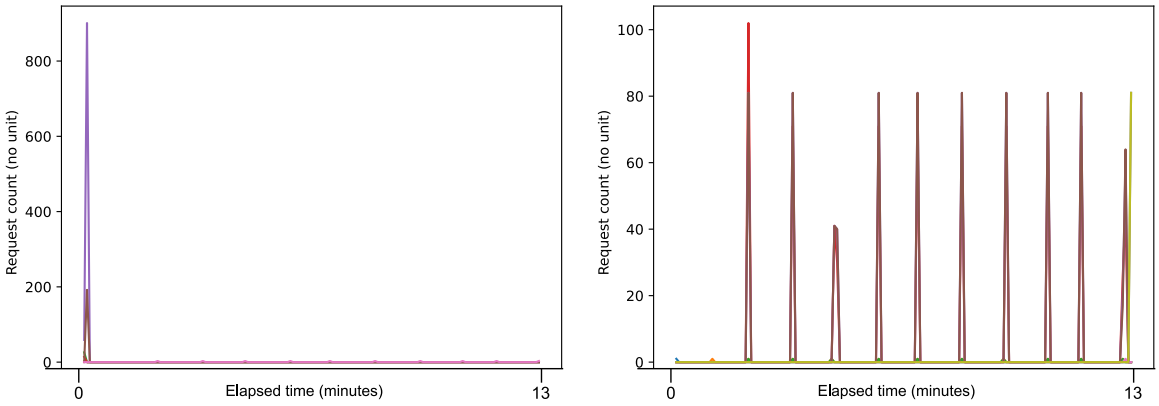


Figure 5: FLC for NAMD application filtered by read/write FLC_events

The findings detailed in Table 6 offer a nuanced perspective on the predictive performance of our algorithm and its comparative analysis with the ARIMA model in the specific context of the NAMD application's READ and WRITE operations. Noteworthy is the perfect accuracy achieved in predicting READ activities, with both Euclidean distance and DTW distance variants, each exhibiting a 100% accuracy, precision, and MCC. This alignment with the ARIMA results emphasizes the reliability of our approach in accurately forecasting read activities, particularly in scenarios where read data is concentrated at the onset of the application. In contrast, predicting WRITE operations presents a more intricate challenge due to the dynamic nature of a 5GB distributed write dataset throughout the application's duration. Here, both Euclidean and DTW distances showcase slightly lower accuracy, precision, and MCC values. Specifically, Euclidean distance achieves an accuracy of 81%, precision of 60%, and MCC of 47%, while DTW distance demonstrates an accuracy of 89%, precision of 76%, and MCC of 68%. In comparison, ARIMA outperforms our approach in predicting WRITE activities, achieving an accuracy of 95%, precision of 88%, and MCC of 0.85. This superior accuracy with ARIMA, however, comes at the cost of considerably higher computational time.

These metrics highlights the strengths and challenges of our approach in predicting distinct file access behaviors within the NAMD application. The impressive accuracy in forecasting READ activities is indicative of the algorithm's capability to capture patterns in a more predictable context. Conversely, the slightly lower performance in predicting WRITE activities underscores the complexities involved in modeling dynamic access patterns. The comparative analysis with ARIMA provides valuable context, showcasing trade-offs between accuracy and computational efficiency in real-world predictive modeling scenarios.

7.2 NEMO results

The results in table 6 show that NEMO's I/O patterns can be predicted with an accuracy 77% on READ and 68% on write using both Euclidean and DTW distances. This quality of prediction correlation is likely due to the fact that NEMO is a data-intensive application that frequently accesses the same data files in a similar way. In the case of WRITE operations, the accuracy, precision, and MCC are slightly better with the Euclidian distance than with DTW's despite its lower calculation time, while ARIMA surpasses both reaching 79% accuracy. For write operations, the fact that the Euclidean distance

Table 5: NAMD evaluation

	Metrics	NAMD		
		Accuracy	Precision	MCC
READ	Euclidean distance	100 %	100 %	1
	DTW distance	100%	100 %	1
	ARIMA	100%	100%	1
WRITE	Euclidean distance	81%	60 %	0.47
	DTW distance	89 %	76 %	0.68
	ARIMA	95%	88%	0.85

(a) NEMO READ Behavior

(b) NEMO WRITE Behavior.

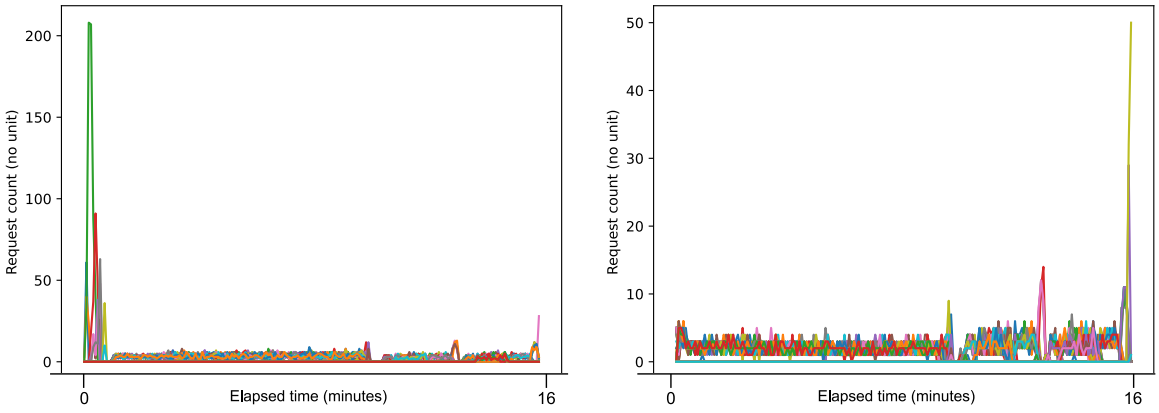


Figure 6: FLC for NEMO application filtered by read/write FLC_events

produces better results than DTW suggests that the patterns do not undergo significant expansion or contraction that would provide an advantage to DTW. We also notice that such as for NEMO case, ARIMA with 72% accuracy does not really outperforms our solution. The results also highlight the importance of custom file placement strategies for applications like NEMO, where file reuse is a significant factor. The predicted I/O patterns can be used to inform file placement decisions, such as placing frequently accessed files in the most efficient storage level using data pre-fetching or data movement. Overall, the high prediction accuracy and the insights gained from analyzing NEMO's I/O patterns provide valuable information for improving the performance.

Table 6: Nemo evaluation

	Metrics	NEMO		
		Accuracy	Precision	MCC
READ	Euclidean distance	77 %	53%	0.37
	DTW distance	77 %	53%	0.37
	ARIMA	79%	57%	0.44
WRITE	Euclidean distance	68%	36 %	0.14
	DTW distance	66%	33 %	0.11
	ARIMA	72%	45%	0.27

(a) LQCD READ Behavior.

(b) LQCD WRITE Behavior.

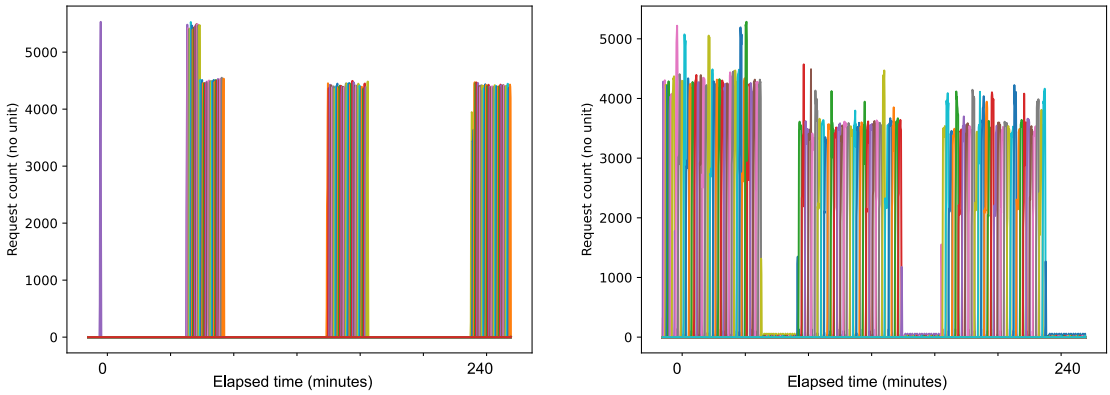


Figure 7: FLC for LQCD application filtered by read/write FLC_events

7.3 LQCD results

In the evaluation of Lattice Quantum Chromodynamics (LQCD), it is notable that unlike the NAMD application, both read (65GB) and write (50GB) operations exhibit comparable levels of activity. This uniformity suggests a balanced workload throughout the entire execution of the LQCD application.

The overall performance of the prediction models in LQCD is noteworthy, with high accuracy rates of 96% for both write and read operations. This underscores the efficacy of the models in capturing intricate patterns within the LQCD application's behavior. A detailed comparison with the ARIMA model reveals interesting insights. In the case of write prediction, Dynamic Time Warping (DTW) marginally outperforms Euclidean Distance (ED) in terms of accuracy and precision. However, ARIMA surpasses both, achieving an impressive 97% accuracy. This signifies the particular effectiveness of ARIMA in capturing the temporal dynamics inherent in the write operations of the LQCD application. Matthews Correlation Coefficient (MCC) values further accentuates the performance metrics. Notably, in the write cases, ARIMA stands out with a higher MCC of 0.93, indicating robust overall performance and solidifying its position as a reliable predictor in the LQCD context. The observed slight superiority of DTW over ED suggests potential variability in the patterns during the execution of the LQCD application.

Table 7: LQCD evaluation

		LQCD		
	Metrics	Accuracy	Precision	MCC
READ	Euclidean distance	96 %	92 %	0.89
	DTW distance	96%	92 %	0.89
	ARIMA	95%	91%	0.88
WRITE	Euclidean distance	95%	91 %	0.89
	DTW distance	96 %	92 %	0.89
	ARIMA	97%	94%	0.93

7.4 Benchmark application (BenchApp) results

The FLCs filtered by READ, as depicted in Figure 8a, reveal distinct patterns that broaden over time. This observation implies that the behavior of files evolves or varies as they are utilized. The dynamic nature of I/O phases, expanding or contracting as shown in Figure 8a, leads to the conclusion that the DTW distance outperforms the Euclidean distance (ED) in this scenario. Specifically, the DTW distance reaches an 89% accuracy compared to 87% for the ED. The precision of DTW is

(a) Bench READ Behavior (b) Bench WRITE Behavior

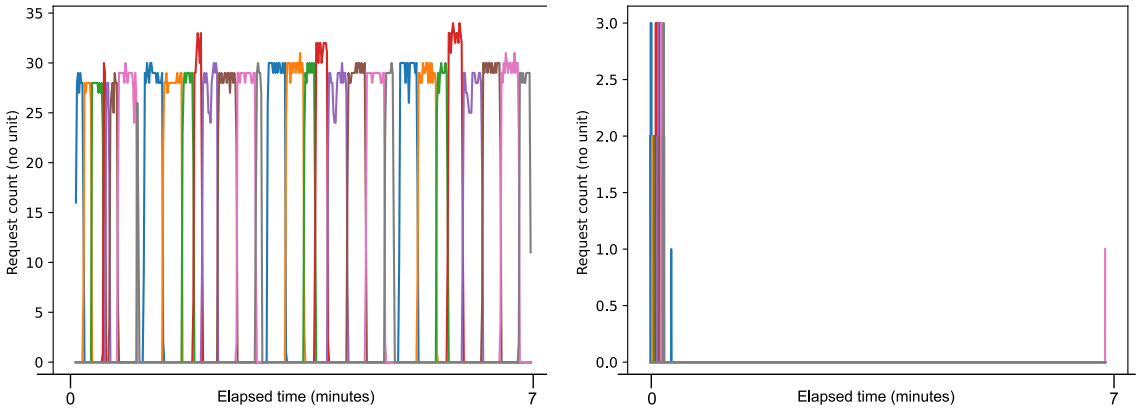


Figure 8: FLC for the benchmark application filtered by read/write FLC_events

75%, surpassing the 71% precision achieved by the ED. This higher performance is attributed to DTW's design, which allows it to adapt to patterns with varying lengths, making it more suitable for scenarios where I/O phases exhibit temporal variations. The adaptability of DTW to align and compare sequences with varying lengths proves advantageous when dealing with time series data characterized by variable patterns or temporal shifts. The results in Table 8 further emphasize the superiority of DTW over the Euclidean distance, showcasing its higher accuracy and precision. Note also, that the DTW distances reaches a 0.68 MCC revealing a relavely high correlation between prediction and ground truth. Furthermore, the comparison with the ARIMA model reveals that DTW maintains a competitive edge. In the READ scenario, DTW outperforms ARIMA in accuracy (89% vs. 87%), despite of a lower precision (75% vs. 76%) and MCC (0.67 vs. 0.68). In the WRITE scenario, both DTW and ARIMA achieve perfect accuracy,precision, and MCC. This suggests that DTW is a robust choice, demonstrating superior predictive capabilities compared to both traditional Euclidean distance metrics and the ARIMA model in this specific application context.

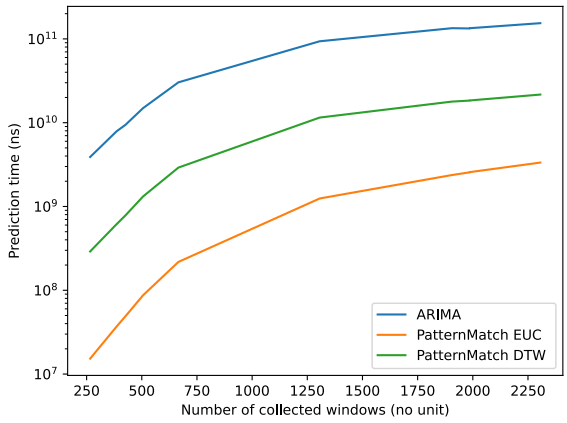
Table 8: Benchmarking App evaluation

		Benchmarking App		
	Metrics	Accuracy	Precision	MCC
READ	Euclidean distance	87 %	71%	0.65
	DTW distance	89%	75 %	0.67
	ARIMA	87 %	76%	0.68
WRITE	Euclidean distance	100%	100%	1
	DTW distance	100 %	100%	1
	ARIMA	100 %	100%	1

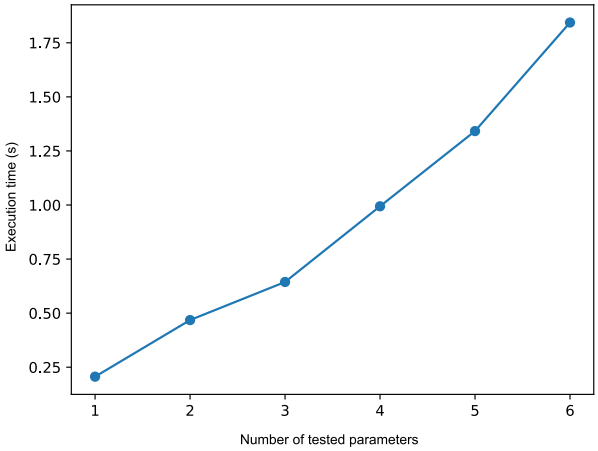
7.5 Prediction time

In figure 9a, we highlight the difference in execution time between the different algorithms, and we can observe that in both cases of Euclidean distance and DTW, the execution remains lower than ARIMA. It is worth noting that the execution times of the differentiation algorithms, required when using ARIMA in the case of non-stationary time serie are not included in these recorded times. Comparing our performance to existing methods in figure 9a, we can see that our algorithm is faster than the reference time serie prediction method (ARIMA) on figure 9a. Additionally, using the Euclidean distance metric reduces the execution time significantly compared to using the DTW distance metric.

Because our window has a fixed size, we see no major improvement in performance between DTW and ED, and we recommend using ED for the associated gain of time when the window has a fixed size. This comparison of execution time do not include the additional steps involved in ARIMA, such as tuning that increases linearly with the number of tested combinations as we can see in figure 9b, stationarity checks, and eventually stationnarisation and destasionnarisation which are variable but non-negligible costs. These steps can be computationally expensive and time-consuming. In contrast, neighrest neighbor prediction may not require these additional preprocessing steps, making it potentially faster overall.



(a) Execution time of the algorithm as a function of the size of the time serie



(b) Execution time of the tuning algorithm

7.6 Hit ratio and anticipated data placement

We compare here the hit ratio on the highest level of the storage hierarchy of the *Burst Buffer* using our File-based migration strategy (F-LRU) against the literature data placement strategies: LRU, LFU, LIRS, ARC and also to FR-LRU. All of them were simulated using our developed tool. We considered the three scenarios as detailed in table 4 with three different configurations for the hierarchy of the *Burst Buffer*. We run the simulation using as input the traces generated by the four aforementioned applications. For comparison purpose, we generated a baseline scenario where the capacity of the highest tier is large enough to capture all the references. So, the hit ratio is the maximum and reaches its upper bound of each application under the assumption of an infinite highest tier. We normalize the number of hits on each tier by this maximum number of hits to obtain the percentage of successful hits out of the possible hits in an ideal configuration with an infinite first tier. In our hit rate table, levels 1 and 2 are interdependent. If we don't achieve success at level 1, we then check for success at level 2, and vice versa. These levels are complementary, meaning that the absence of success in one level prompts an assessment in the other to gauge overall performance. However, it's crucial to understand that if we attain a higher hit rate at level 1, it is sufficient to conclude that the strategy is better. Success at level 1 takes precedence in this evaluation, and superior performance at this level alone indicates the overall effectiveness of the strategy. On the other hand, a lower hit rate at level 2, when success is already achieved at level 1, may not significantly impact the assessment, as the primary focus is on the success achieved at the more critical level 1.

7.6.1 Small configuration

We can see on table 9 that the number of hits is improved using our strategy for both NEMO and the synthetic benchmark application we developed. Specifically for NEMO, our File-based strategy (F-LRU) outperforms all the tested strategies with a better hit rate at the highest level of the hierarchy with the lowest latency. Note that, LFU and ARC performs better than LRU for the case of NEMO but conversely LRU takes the advantage in the case of NAMD. This is justified by the fact that in

such a configuration, recently accessed blocks are more reused than frequently ones for NAMD and vice versa for NEMO. We observe also that the suitability of LRU or LFU is more dependent of the application whereas our strategy remains stable and offers a better hit rate for both NEMO and BenchApp. We also note the under-performance of LIRS and ARC strategies, in the case of LQCD that can be attributed to their inability to adapt efficiently to large-scale data movements. These strategies decide which block to remove based on the state of their stacks at a given point. However, the challenge arises when multiple blocks are removed simultaneously without allowing the stacks to adapt adequately to the changes. This lack of synchronization hinders the strategies ability to make optimal replacement decisions during substantial data transfers. F-LRU consistently delivers better results due to its consideration of not only block recency or frequency but also higher-level contextual information through the file reuse property.

Table 9: F-LRU hit rates compared to ARC LIRS LRU and LFU as (level1, level2) couples for the small configuration

	Politiques					
	LRU	LFU	ARC	LIRS	FR-LRU	F-LRU
NEMO	41%, 57%	43%, 54%	43%, 55%	41%, 56%	46%, 52%	47%, 52%
LQCD	41%, 0%	41%, 0%	34%, 6%	4%, 14%	32%, 22%	41%, 0%
NAMD	87%, 12%	88%, 11%	88%, 11%	52%, 25%	77%, 23%	87%, 12%
BenchApp	0%, 1%	0%, 1%	0%, 1%	0%, 1%	0%, 1%	0%, 4%

7.6.2 Intermediate configuration

This configuration table 10 corresponds to a larger capacity for both the two highest tiers of the *burst Buffer* to represent a configuration with less restrictions on resources compared to the first scenario. In this case, we can observe a significant change for NEMO which idealy performs for the three strategies because the capacity of the highest tier is large enough to accommodate more data, resulting in a higher hit rate (98% to 100%). On the contrary, for LQCD, we observe that LRU and F-LRU, with a hit rate of 97%, outperform LFU, which has a hit rate of only 50% because in this configuration LQCD recently accessed blocks are more reaccessed than frequently ones. We also note that ARC makes a better result than LFU with a 72% contrary to LIRS which only reaches 29% on the highest tier. It should also be noted that FR-LRU by its randomness gives low results (73%) compared to F-LRU prediction approach 96%. Regarding NAMD, all its data fit in the highest tier leading to a 100% hit rate. Finally, for the synthetic benchmark, our F-LRU strategy performs better than the other ones with a hit rate of 75% against 71% for all of LRU, LFU,FR-LRU and ARC and 67% for LIRS. So, F-LRU performs at least as well as the best of LRU and LFU for all applications.

Table 10: FR-LRU and F-LRU hit rates compared to ARC LIRS LRU and LFU as (level1, level2) couples for the intermediate configuration

	Politiques					
	LRU	LFU	ARC	LIRS	FR-LRU	F-LRU
NEMO	99%, 1%	98%, 2%	98%, 2%	98%, 2%	99%, 1%	100%, 0%
LQCD	96%, 4%	50%, 50%	72%, 28%	29%, 57%	73%, 26%	96%, 4%
NAMD	100%, 0%	100%, 0%	100%, 0%	100%, 0%	100%, 0%	100%, 0%
BenchApp	71%, 29%	71%, 29%	71%, 29%	67%, 26%	71%, 29%	75%, 25%

7.6.3 Large configuration

The configuration displayed in table 11 represents a configuration almost without restrictions on resources as the tiers storage capacities are significantly larger than the precedent scenarios. Consequently, the highest tier can capture all theI/Otraffic rising its hit rate to 100% and no data migration is needed. Let's note that within the total of 100%, cold access is not included. These accesses can only be hit through a prefetch mechanism.

Table 11: FR-LRU and F-LRU hit rates compared to ARC LIRS LRU and LFU as (level1, level2) couples for the large configuration

	Politiques					
	LRU	LFU	ARC	LIRS	FR-LRU	F-LRU
NEMO	100%, 0%	100%, 0%	100%, 0%	100%, 0%	100%, 0%	100%, 0%
LQCD	100%, 0%	100%, 0%	100%, 0%	100%, 0%	100%, 0%	100%, 0%
NAMD	100%, 0%	100%, 0%	100%, 0%	100%, 0%	100%, 0%	100%, 0%
BenchApp	100%, 0%	100%, 0%	100%, 0%	100%, 0%	100%, 0%	100%, 0%

7.7 Read Volumes

We have now examined the impact of these hit rates on the read volumes made on the most performant tier (RAM). For this purpose, we have generated bar plots ranging from figure 10 to 13 for the first two configurations listed in Table 4. The last one not providing additional information. The y-axis on a logarithmic scale enables a relative comparison, with the first configuration represented in blue and the second in red. The height of each bar indicates the number of 4K blocks read from the RAM. On the x-axis, the various cache management policies are displayed. Note that the volume in Gb could be get by the formula :

$$Volume(Gb) = \frac{\text{block\_number} \times 4096}{1024^3} \tag{3}$$

We first examine the case of NAMD, where notable differences among cache management policies are not considerable. Even in the scenario where all queries result in hits (intermediate configuration), 1695 blocks totaling 8MB, this figure is significantly lower than the 2GB of reads executed by the application (see Table 2). This highlights the importance of cold accesses and suggests that a downward migration strategy alone is insufficient. Only prefetching appears as a solution for enhancing I/O latency. It is noteworthy that our strategy demonstrates competitiveness with state-of-the-art approaches, as evidenced by bars that are relatively close to others in both configurations.

In the case of NEMO, the scenario differs significantly, as the application exhibits substantial cold access, amounting to 60GB, of which 44GB are reused. Notably, in the first configuration, there is a noticeable increase in the volume, reaching 5,488,205 blocks equivalent to 21GB. This surpasses traditional policies such as LRU and LFU, which manage 18GB, and even ARC, which handles 19GB. This outcome emphasizes the efficacy of our cache management strategy, particularly in comparison to conventional approaches, in efficiently handling the specific access patterns of the NEMO application.

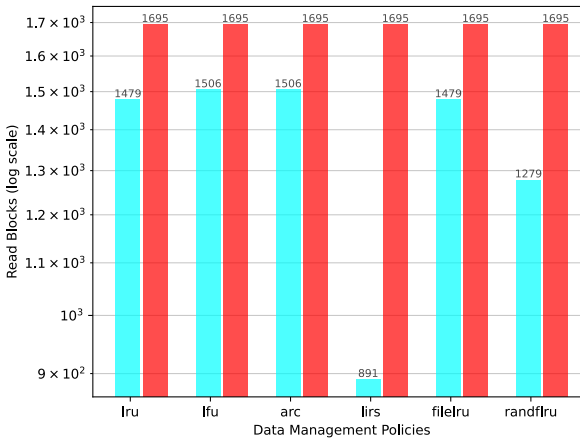


Figure 10: NAMD

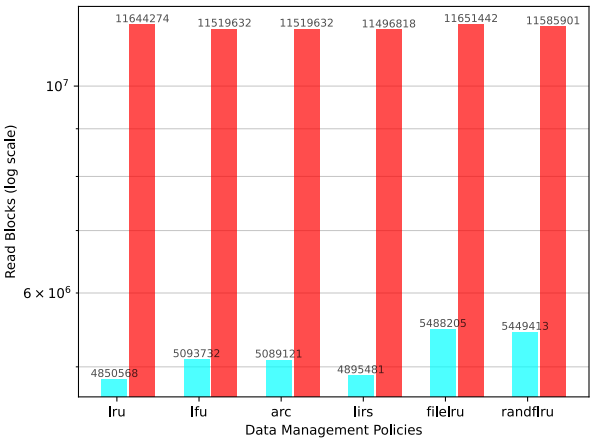


Figure 11: NEMO

In the case of LQCD, our cache management strategy demonstrates performance comparable to the best outcomes achieved by standard strategies in the first configuration, and it slightly outperforms them in the second configuration. Specifically, the volume of re-used blocks amounts to 45GB, whereas

the application initiates 42GB of read operations. This is attributed to a significant level of data reuse, where the amount of re-use can surpass the quantity of generated reads. This occurrence is associated with the fact that accessing a subset of a block results in the loading of the entire block. This behavior is directly influenced by the file management approach detailed in Section 6.5. Regarding the Bench application, in the first configuration, it appears that our strategy performs less effectively than other strategies such as LRU, LFU, and LIRS with 71Mb against respectively 75Mb, 142 Mb, and 273Mb. However, this may not be representative as it concerns less than 0% of the possible hits, as indicated in Table 9. On the other hand, for the second configuration, a notable improvement is observed with our strategy achieving 22GB at the file level, surpassing both ARC and LFU, which achieve 21GB and outperform all other policies.

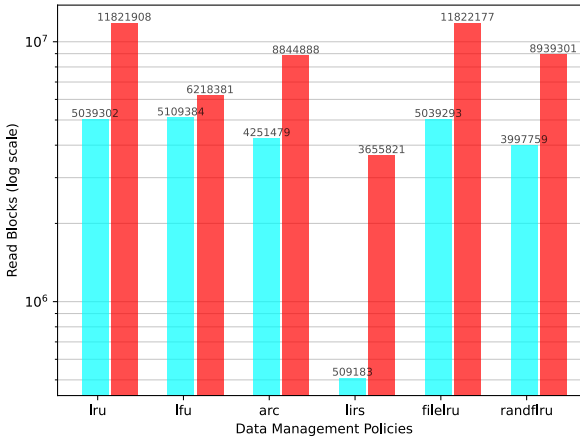


Figure 12: LQCD

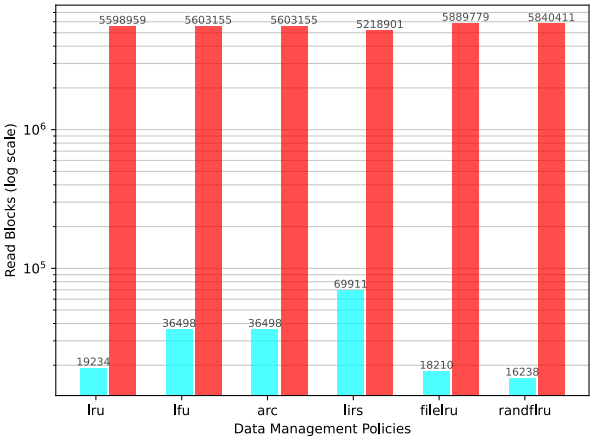


Figure 13: BenchApp

8 Conclusion and further works

In this work, we shifted from a block-based approach to a file-based approach that shows great promise in accurately predicting I/O patterns, especially in terms of anticipating file reuse probabilities: we have shown that in terms of accuracy, the file-based approach achieves result ranging from 77% accuracy and 53% precision in the most difficult scenarios, to a perfect 100% accuracy and precision in cases where file behavior is easier to predict. Whenever comparing cache hits, the results we obtained are promising, demonstrating that our proposed F-LRU (File-based Least Recently Used) is at least as effective as effective as LRU, with a notable 1.06-fold increase in hit rate specifically in the context of the IO-Bench application. Furthermore, it also shows better results than LFU, ARC and LIRS, increasing the hit rate by a factor of 1.94 compared to LFU, 1.33 against ARC for LQCD, 3.31 factor against LIRS, and 1.06 against LRU in the context of LQCD application.
The experimental evaluation results demonstrate that our algorithm achieves high prediction accuracy for file-level I/O patterns compared to existing techniques. Additionally, the algorithm's complexity is competitive with state-of-the-art methods, making it a practical and efficient solution for real-world scenarios.
In this study, our primary objective was to predict the order of priority in data access. We initially focused on a specific application, namely data replacement, as an illustration of our predictive approach. However, it's crucial to clarify that our developed ranking method is inherently versatile and extends beyond data management. It could have just as easily been applied to other applications, such as prefetching. Our research opens avenues for future exploration, where we plan to extend the application of our predictive approach to various scenarios, including but not limited to other other strategies like prefetching. In this study, our focus has predominantly centered on an intra-file approach, analyzing patterns within individual files. However, looking ahead, we are considering the incorporation of an inter-file approach. This involves merging patterns from different files and calculating the weighted average of the nearest neighbors, a technique demonstrated to yield promising results in the literature [28] and enhance data placement strategies. The idea behind exploring an inter-file approach lies in the

recognition that files within a system often exhibit similarities in their access patterns. By extending our analysis to include both intra-file and inter-file comparisons, we aim to gain a more comprehensive understanding of data access behaviors. This holistic perspective is anticipated to contribute to improve the prediction accuracy by capturing patterns of similarity not only within files but also between different files. For example, files for inputs, outputs, or checkpoints may belong to specific categories and exhibit similar behaviors. Analyzing these categories allows us to observe recurrent patterns that transcend the individual boundaries of files. Including these inter-file comparisons would capture similarities in behavior between different file categories, contributing to a more precise and holistic prediction of data accesses.

9 Data Availability statement

The data underlying this article will be shared with the corresponding author upon reasonable request.

References

[1] Sbb Flash Accelerator. https://atos.net/en/2019/product-news_2019_02_07/atos-boosts-hpc-application-efficiency-new-flash-accelerator-solution.

[2] Zaher Al Aghbari and Ayoub Al-Hamadi. Finding k most significant motifs in big time series data. *Procedia Computer Science*, 170:595–601, 2020. DOI: 10.1016/j.procs.2020.03.131.

[3] Dima Alberg and Mark Last. Short-term load forecasting in smart meters with sliding window-based arima algorithms. *Vietnam Journal of Computer Science*, 5:241–249, 2018.

[4] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 513–526, 2020. DOI: 10.1145/3373376.3378498.

[5] Sorav Bansal and Dharmendra S Modha. CAR: Clock with Adaptive Replacement. *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, page 15, 2004.

[6] Francieli Zanon Boito, Ramon Nou, Laércio Lima Pilla, Jean Luca Bez, Jean-François Méhaut, Toni Cortes, and Philippe OA Navaux. On server-side file access pattern matching. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 217–224. IEEE, 2019. DOI:10.1109/HPCS48598.2019.9188092.

[7] Christine TH Davies, E Follana, A Gray, GP Lepage, Q Mason, M Nobes, J Shigemitsu, HD Trottier, M Wingate, C Aubin, et al. High-precision lattice qcd confronts experiment. *Physical Review Letters*, 92(2):022001, 2004. DOI :10.1103/PhysRevLett.92.022001.

[8] K Lalitha Devi and S Valli. Time series-based workload prediction using the statistical hybrid model for the cloud environment. *Computing*, 105(2):353–374, 2023. DOI: 10.1007/s00607-022-01129-7.

[9] Bin Dong, Xiuqiao Li, Qimeng Wu, Limin Xiao, and Li Ruan. A dynamic and adaptive load balancing strategy for parallel file system with large-scale i/o servers. *Journal of Parallel and distributed computing*, 72(10):1254–1268, 2012. DOI: 10.1016/j.jpdc.2012.05.006.

[10] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, December 2001. DOI: 10.1109/TC.2001.970573.

[11] Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, and Rob Ross. Omnisc'io: a grammar-based approach to spatial and temporal i/o patterns prediction. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 623–634. IEEE, 2014. DOI :10.1109/SC.2014.56.

[12] Ohad Eytan, Danny Harnik, Effi Ofer, and Roy Friedman. It's Time to Revisit LRU vs. FIFO. *HotStorage'20: Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File System*, page 7, 2020.

[13] San Francisco. Proceedings of FAST '03: 2nd USENIX Conference on File and Storage Technologies. *Adaptive Replacement Cache (ARC)*, page 17, 2023.

[14] Felix Freitag, Julita Corbalan, and Jesus Labarta. A dynamic periodicity detector: Application to speedup computation. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 6–pp. IEEE, 2001. DOI: DOI:10.1109/IPDPS.2001.924928.

[15] Bryan Harris and Nihat Altiparmak. Ultra-low latency ssds impact on overall energy efficiency. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[16] Song Jiang and Xiaodong Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. *ACM SIGMETRICS Performance Evaluation Review*, page 12, 2002. DOI: 10.1145/511399.511340.

[17] Rohit J Kate. Using dynamic time warping distances as features for improved time series classification. *Data Mining and Knowledge Discovery*, 30:283–312, 2016.

[18] Adrian Khelili, Sophie Robert Hayek, and Soraya Zertal. Forecasting file lifecycles for intelligent data placement in hierarchical storage. In *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 181–191. IEEE Computer Society, 2023.

[19] Adrian Khelili, Sophie Robert, and Soraya Zertal. Filip: A file lifecycle-based profiler for hierarchical storage. *INFOCOMMUNICATIONS JOURNAL*, 14(4):26–33, 2022. DOI: 10.36244/ICJ.2022.4.4.

[20] Julian Kunkel, Michaela Zimmer, and Eugen Betke. Predicting performance of non-contiguous i/o with machine learning. In *High Performance Computing: 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings 30*, pages 257–273. Springer, 2015. DOI: 10.1007/978-3-319-20119-1_19.

[21] Jessica Lin, Eamonn Keogh, Li Wei, and Stefano Lonardi. Experiencing sax: a novel symbolic representation of time series. *Data Mining and knowledge discovery*, 15:107–144, 2007.

[22] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2012.

[23] Jakob Lüttgau, Michael Kuhn, Kira Duwe, Yevhen Alforov, Eugen Betke, Julian Kunkel, and Thomas Ludwig. Survey of storage systems for high-performance computing. *Supercomputing Frontiers and Innovations*, 5(1), 2018.

[24] Francisco Martínez, María Pilar Frías, María Dolores Pérez, and Antonio Jesús Rivera. A methodology for applying k-nearest neighbor to time series forecasting. *Artificial Intelligence Review*, 52(3):2019–2037, 2019.

[25] Dhruv Matani, Ketan Shah, and Anirban Mitra. An O(1) algorithm for implementing the LFU cache eviction scheme. Technical Report arXiv:2110.11602, arXiv, October 2021. DOI: 10.48550/arXiv.2110.11602.

[26] Ryan McKenna, Stephen Herbein, Adam Moody, Todd Gamblin, and Michela Taufer. Machine learning predictions of runtime and io traffic on high-end clusters. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 255–258. IEEE, 2016. DOI:10.1109/CLUSTER.2016.58.

[27] James Oly and Daniel A Reed. Markov model prediction of i/o requests for scientific applications. In *Proceedings of the 16th international conference on Supercomputing*, pages 147–155, 2002. DOI: 10.1145/514191.514214.

[28] R Keith Oswald, William T Scherer, and Brian L Smith. Traffic flow forecasting using approximate nearest neighbor nonparametric regression. Technical report, 2000.

[29] 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.

[30] Liana V Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, and Giri Narasimhan. Learning Cache Replacement with Cacheus. *FAST '21*, page 15, 2021.

[31] F. Sevault, S. Somot, and J. Beuvier. A regional version of the NEMO ocean engine on the Mediterranean Sea: NEMOMED8 user's guide. Technical report, Meteo-France, CNRM, 2009.

[32] Stephan Spiegel. Time series distance measures. *Ph.D. thesis. Berlin, Germany.*, 2015.

[33] Mathieu Stoffel, François Broquedis, Frédéric Desprez, and Abdelhafid Mazouz. Phase-ta: Periodicity detection and characterization for hpc applications. In *HPCS 2020-18th IEEE International Conference on High Performance Computing and Simulation*, pages 1–12. IEEE, 2021.

[34] Pradeep Subedi, Philip Davis, Shaohua Duan, Scott Klasky, Hemanth Kolla, and Manish Parashar. Stacker: an autonomic data movement engine for extreme-scale data staging-based in-situ workflows. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 920–930. IEEE, 2018.

[35] Andrew S. Tanenbaum. *Modern operating systems*. Pearson, Boston, fourth edition edition, 2015.

[36] Houjun Tang, Xiaocheng Zou, John Jenkins, David A Boyuka, Stephen Ranshous, Dries Kimpe, Scott Klasky, and Nagiza F Samatova. Improving read performance with online access pattern analysis and prefetching. In *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings 20*, pages 246–257. Springer, 2014. DOI : 10.1007/978-3-319-09873-9_21.

[37] Nancy Tran and Daniel A Reed. Arima time series modeling and forecasting for adaptive i/o prefetching. In *Proceedings of the 15th international conference on Supercomputing*, pages 473–485, 2001. DOI: 10.1145/377792.377905.

[38] Marc-André Vef, Nafiseh Moti, Tim Süß, Markus Tacke, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. Gekkofs—a temporary burst buffer file system for hpc applications. *Journal of Computer Science and Technology*, 35:72–91, 2020.

[39] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving Cache Replacement with ML-based LeCaR. *HotStorage'18: Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems*, page 6, 2018.

[40] Lipeng Wan, Matthew Wolf, Feiyi Wang, Jong Youl Choi, George Ostrouchov, and Scott Klasky. Analysis and modeling of the end-to-end i/o performance on olcf's titan supercomputer. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1–9. IEEE, 2017. DOI: 10.1109/HPCC-SmartCity-DSS.2017.1.

[41] Xiaoqian Wang, Yanfei Kang, Rob J Hyndman, and Feng Li. Distributed arima models for ultra-long time series. *International Journal of Forecasting*, 2022. DOI : 10.1016/j.ijforecast.2022.05.001.

[42] Herman Wold. *A study in the analysis of stationary time series*. PhD thesis, Almqvist & Wiksell, 1938.