



HAL
open science

Capturing Periodic I/O Using Frequency Techniques

Ahmad Tarraf, Alexis Bandet, Francieli Zanon Boito, Guillaume Pallez, Felix Wolf

► **To cite this version:**

Ahmad Tarraf, Alexis Bandet, Francieli Zanon Boito, Guillaume Pallez, Felix Wolf. Capturing Periodic I/O Using Frequency Techniques. IPDPS 2024 - 38th IEEE International Parallel & Distributed Processing Symposium, May 2024, San Francisco, United States. pp.1-13. hal-04382142

HAL Id: hal-04382142

<https://inria.hal.science/hal-04382142v1>

Submitted on 2 Mar 2024


HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Capturing Periodic I/O Using Frequency Techniques

Ahmad Tarraf 

Department of Computer Science
Technical University of Darmstadt
Darmstadt, Germany
ahmad.tarraf@tu-darmstadt.de

Alexis Bandet 

Univ. Bordeaux, CNRS, Bordeaux INP
Inria, LaBRI, UMR 5800
Talence, France
alexis.bandet@inria.fr

Francieli Boito 

Univ. Bordeaux, CNRS, Bordeaux INP
Inria, LaBRI, UMR 5800
Talence, France
francieli.zanon-boito@u-bordeaux.fr

Guillaume Pallez 

Inria
Rennes, France
guillaume.pallez@inria.fr

Felix Wolf 

Department of Computer Science
Technical University of Darmstadt
Darmstadt, Germany
felix.wolf@tu-darmstadt.de

Abstract—Many HPC applications perform their I/O in bursts that follow a periodic pattern. This allows for making predictions as to when a burst occurs. System providers can take advantage of such knowledge to reduce file-system contention by actively scheduling I/O bandwidth. The effectiveness of this approach, however, depends on the ability to *detect* and *quantify* the periodicity of I/O patterns online. In this paper, we introduce FTIO, an online method to detect periodic I/O phases, which is based on discrete Fourier transform (DFT), combined with outlier detection. We provide metrics that gauge the confidence in the output and tell how far from being periodic the signal is. We validate our approach with large-scale experiments on a production system and examine its limitations extensively. Our experiments show that FTIO has a mean error below 11%. Finally, we demonstrate that FTIO allowed the I/O scheduler Set-10 to boost system utilization by 26% and reduce I/O slowdown by 56%.

Index Terms—HPC, I/O prediction, temporal I/O behavior

I. INTRODUCTION

HPC applications often alternate between compute phases and access to storage [1]–[3]. Common practices in HPC, such as checkpointing or visualization [3], [4], make the I/O phases often periodic, and usually involve long file system accesses,

This work was funded by the European Commission and the German Federal Ministry of Education and Research (BMBF) under the EuroHPC programmes DEEP-SEA (GA no. 955606, BMBF funding no. 16HPC015) and ADMIRE (GA no. 956748, BMBF funding no. 16HPC006K), which receive support from the European Union’s Horizon 2020 programme and DE, FR, ES, GR, BE, SE, GB, CH (DEEP-SEA) or DE, FR, ES, IT, PL, SE (ADMIRE). Furthermore, we express our gratitude toward the French National Research Agency (ANR) for the financial support in the frame of DASH (ANR-17-CE25-0004), by the Project Région Nouvelle Aquitaine 2018-1R50119 “HPC scalable ecosystem”. Moreover, this work received further funding from the German Federal Ministry of Education and Research (BMBF) and the Hessian Ministry of Science and Research, Art and Culture (HMWK) as part of the NHR Program. The authors gratefully acknowledge the computing time provided to them on the high-performance computer Lichtenberg at the NHR Center NHR4CES@TUDa. This is funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Science and Research, Art and Culture (HMWK). Computations for this research were performed using computing resources under Project 2215. Finally, the authors also gratefully acknowledge the computing time provided to them on the PlaFRIM experimental testbed supported by INRIA, CNRS (LaBRI and IMB), Université de Bordeaux, Bordeaux INP and *Conseil Régional d’Aquitaine* (see <https://www.plafrim.fr>).

which can be a source for I/O and network contention. Aside from causing performance variability [5], [6], contention means that jobs run longer, harming the platform’s utilization and ultimately wasting resources. Solutions proposed to alleviate these aspects include I/O scheduling [3], [7]–[10], I/O-aware batch scheduling [11]–[14], and the use of burst buffers [15]–[17]. A challenge when designing such solutions is obtaining knowledge of the applications’ I/O patterns.

There are several approaches to gathering I/O knowledge depending on the precision needed. The most popular tool is probably Darshan [18], which gathers aggregated metrics. However, these aggregated metrics do not properly represent the *temporal* behavior of applications [19]. Because I/O tends to be bursty and periodic, knowing how many bytes are accessed does not paint the full picture. We need to know *when* (or rather *how often*) these accesses happen: two applications that write each 1 TB over 2 hours, one with a single I/O phase at the end of the execution and the other one with multiple I/O phases every 2 minutes, impose very different loads on the system. Thus, one might desire a detailed description of I/O activity over time. However, finding an extremely precise profile comes at the cost of a higher overhead, both in terms of measurement and data accumulation. Moreover, a detailed time model can be hard to explore in a contention-avoidance algorithm that is lightweight enough to be used in practice, especially as models with high predictive accuracy are often *black box* and cannot be interpreted directly for explaining I/O performance [20]. Hence, depending on the use case, models at higher abstraction levels might be tolerated, which can be easier to interpret and often to generate, especially online.

Recent work on I/O scheduling [7], [9], [10], [21] has shown that knowledge of periodic I/O patterns, even when not *perfectly precise*, leads to good contention avoidance. Consequently, one approach could be to predict the *period of the I/O phases* during runtime and provide this information to such approaches rather than finding detailed time models. This is the metric we seek in this work, which presents a trade-off between aggregated information and a detailed time model.

However, describing the temporal I/O behavior in terms of I/O phases is a challenging task. Indeed, the HPC I/O stack only sees a stream of issued requests and does not provide I/O behavior characterization. Contrary, the notion of an *I/O phase* is often purely logical, as it may consist of a set of independent I/O requests, issued by one or more processes and threads during a particular time window, and popular APIs do not require that applications explicitly group them. Thus, a major challenge is to draw the borders of an *I/O phase* (see Figure 1). Consider, for example, an application with 10

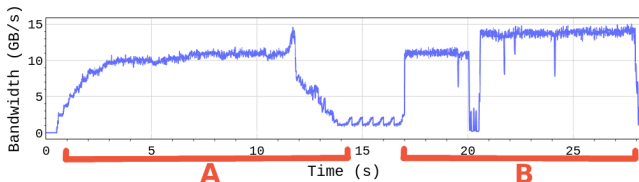


Fig. 1: Difficulty of detecting I/O phases: Where does A finish? Is B one or two phases? Why don't A and B belong together?

processes that writes 10 GB by generating a sequence of two 512 MB write requests per process, then performs computation and communication for a certain amount of time, after which it writes again 10 GB. How do we assert that the first 20 requests correspond to the first I/O phase and the last 20 to a second one? An intuitive approach is to compare the time between consecutive requests with a given threshold to determine whether they belong to the same phase. Naturally, the suitable threshold should depend on the system. The reading or writing method can make this an even more complex challenge, as accesses can occur, e.g., during computational phases in the absence of barriers. Hence, the threshold would not only be *system dependent* but also *application dependent*, making this intuitive approach more complicated than initially expected.

Even assuming that one is able to find the boundaries of various I/O phases, this might still not be enough. Consider, for example, an application that periodically writes large checkpoints with all processes. In addition, a single process writes, at a different frequency, only a few bytes to a small log file. Although both activities clearly constitute I/O, only the period of the checkpoints is relevant to contention-avoidance techniques. If we simply see I/O activity as belonging to I/O phases, we may observe a profile that does not reflect the behavior of interest very well.

Thus, a method for characterizing the temporal I/O behavior of an application is needed that determines the period of the I/O phases and is thus at a higher abstraction level compared to detailed time modeling approaches. To be useful in practice, it should impose minimal overhead and generate only a modest amount of information, especially during the *online execution* of an application. This is where our paper aims to contribute:

- We propose FTIO, which characterizes the temporal I/O behavior of an application in terms of its period, obtained using frequency techniques. Additionally, we provide strategies to adapt to behavioral changes.

- We introduce metrics that quantify the confidence in the obtained results and further characterize the I/O behavior based on the identified period.
- FTIO is implemented as an open-source library, which can be easily attached to existing codes, to provide online predictions of their I/O behavior with low overhead. Moreover, we offer an offline realization as well.
- We evaluate FTIO with large-scale applications and extensively study its limitations and accuracy using traces crafted to represent challenging situations. Moreover, we show how FTIO enables an I/O scheduler to reduce I/O slowdown by 56% and boost system utilization by 26%.

By finding the period of I/O, the average amount of data and time spent per I/O phase can be calculated, which has clear usefulness for burst buffer management, for example.

This paper is organized as follows: In Section II, we present our approach: how we collect information, DFT and how we use it, the additional confidence metrics, and the implementation of FTIO. Our strategy is evaluated in Section III, while in Section IV we illustrate the use of FTIO for I/O scheduling. Finally, we discuss related work in Section V, before concluding on the implications of this work in Section VI.

II. FTIO: FINDING THE PERIOD OF I/O

This section presents our approach to characterize the temporal I/O behavior of an HPC application in terms of the period of its I/O phases. We call our methodology: Frequency Techniques for I/O (FTIO), as it leverages well-known signal-processing techniques. FTIO is implemented as a two-step approach: (1) a library at the application side intercepts the I/O calls and appends the collected data continuously to a file (Section II-A), which (2) can be evaluated at any time (Sections II-B till II-E), on a cluster or a local machine, to determine the period of the I/O phases. Both the tracing library named TMIO¹ (Tracing MPI-IO) and FTIO² are publicly available on GitHub. In what follows, we describe TMIO as part of FTIO.

A. Gathering the I/O Information

As the first step of FTIO, the I/O information from the application needs to be collected. For this, we developed a tracing library in C++ (TMIO) that intercepts specific MPI-IO calls to gather metrics such as start time, end time, and transferred bytes. We provide two methods for linking the library to the application, depending on whether the information is used for offline (*detection*) or online (*prediction*) periodicity analysis. The offline mode uses the LD_PRELOAD mechanism. Upon MPI_Finalize, the collected data is written to a single file to be analyzed later. In the online mode, the application is compiled with our library and a single line is added to indicate when to flush the results out to a file (JSON Lines or MessagePack [22]). This file can be evaluated anytime using a Python script to dynamically predict the period of the next phases based on

¹<https://github.com/tuda-parallel/TMIO/>

²<https://github.com/tuda-parallel/FTIO/>

the data collected up to this point. Note that at the end of the run, the same file can be used for offline evaluation. Our library has low overhead as the I/O data is collected at the rank level (individual requests). The overlapping of the requests (i.e., bandwidth at the application level) is evaluated by the Python script (either entirely or for a given time window) with a linear complexity with the number of I/O requests.

What we need for the next step is essentially the variation of the bandwidth over time. As the analysis is at the application level, we merge the information we collected per process. Note that although we used our library in this paper, it could easily be replaced by other tools and data sources (e.g., file system monitoring data if available) For the *detection* approach, for example, we support Recorder [23] and Darshan [1], [18] profile and traces. The next section describes how the collected information $x(t)$ (bandwidth over time) is further processed.

B. Extracting the Period of I/O

With our approach, we move away from detailed modeling and focus on a simple metric: The period of the I/O phases. For that, we examine the I/O behavior in the frequency domain instead of traditionally analyzing it in the time domain. Thus, we treat the I/O bandwidth over time as a *signal*, which we first discretize and then analyze using the discrete Fourier transform (DFT). Since we aim to find the period of a signal rather than fully model its time behavior, frequency analysis coupled with outlier detection perfectly suits this task. Compared to time analysis, frequency techniques, such as DFT, decompose a signal into its frequency components, giving us control over the *interesting* I/O. Moreover, as we focus on I/O phases rather than individual requests, by applying DFT on the application-level signal, we overcome the challenges from Section I. Additionally, the parameters of DFT allow FTIO to adapt to changing I/O behavior and specify the range of interesting I/O as handled later in Sections II-D and II-E.

1) *DFT*: DFT decomposes a signal into its frequency components such that their sum allows reconstructing the signal. As an input of DFT, the continuous signal $x(t)$ is discretized with a sampling frequency f_s to obtain $N = \Delta t \cdot f_s$ samples:

$$\{x_n = x(n/f_s) \mid n \in [0, N)\}$$

DFT transforms this evenly-spaced sequence from the time domain into a sequence X_k of the same size with $k \in [0, N)$ bins in the frequency domain:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi k n}{N} i},$$

for the frequencies:

$$f_k = \frac{k}{N} f_s = \frac{k}{\Delta t}$$

Thus, DFT is evaluated for the fundamental frequency $\frac{f_s}{N}$ and its harmonics. As consecutive frequencies f_k are spaced $1/\Delta t$ apart, the larger the time window Δt , the closer the components X_k are to each other, thus improving the precision of DFT. However, this increases the complexity of the analysis.

Since the sampled signal x_n consists of purely real values for our purposes (I/O signal), DFT is symmetric and only half of the frequencies are needed to reconstruct the original signal with the inverse DFT (IDFT):

$$x_n = \frac{1}{N} \left(X_0 + \sum_{k=1}^{\frac{N}{2}} 2|X_k| \cos \left(\frac{2\pi k n}{N} + \arg(X_k) \right) \right) \quad (1)$$

with the amplitude $|X_k|$ and the phase $\arg(X_k)$. This reduces the calculation needed for the reconstruction of the signal and limits the constituting signals to cosine waves only, simplifying the interpretation of results. Consequently, when plotting the amplitude $|X_k|$ against the frequencies f_k , only half of the spectrum (*single-sided spectrum*) needs to be inspected. In this case, the amplitude of the fundamental and harmonics need to be multiplied by two, as shown in Eq. (1).

X_0 , the DC offset in signal analysis terminology, is expected to be among the highest components as the I/O data transferred is always a positive number of bytes, and the cosine waves obtained with DFT need to be shifted upwards. Note that, to compute DFT, we use the Fast Fourier Transform (FFT) algorithm, which has a complexity of $O(N \log N)$.

After obtaining the amplitude spectrum from DFT, we need to examine it to extract the period of I/O phases. However, I/O tends to exhibit variations and is often affected by noise, which results in high frequencies with small amplitudes. To alleviate this effect, we use the *power spectrum* ($p_k = \frac{1}{N} X_k^2$) instead of the amplitude spectrum. By normalizing the power spectrum over the total power of the signal for the plots, the y-axis of the normalized power spectrum indicates the *contribution* of the frequency to the total signal power.

2) *Outlier detection*: The most straightforward approach for extracting the period of the I/O phases is to find the frequency with the highest contribution, i.e., the *dominant frequency* f_d , while excluding the DC offset from the analysis. Intuitively, the period is simply then $1/f_d$. However, if there are multiple frequencies with similarly high contributions, the frequency with the maximum contribution does *not* properly represent the temporal behavior. Notably, that is the case for non-periodic signals. Hence, simply selecting the maximum would silently accept a result that is probably inaccurate; it also has to be an *outlier*. One approach for detecting outliers is the Z-score [24]. It reveals how many standard deviations σ a power p_k is from the mean \bar{p} of all powers:

$$z_k = \frac{|p_k| - |\bar{p}|}{\sigma} \quad (2)$$

For each frequency f_k with $k \in [1, \frac{N}{2}]$, a Z-score z_k is found. A Z-score beyond 3 usually indicates an outlier [24]. As there might be several outliers, to find the dominant frequencies, we compare the Z-scores of the outliers to the largest Z-score $z_{\max} = \max_{k \geq 1} (z_k)$. Moreover, if a frequency f_k is an outlier ($z_k > 3$), and its Z-score is within 80% of the largest Z-score (a tolerance value that can be adjusted) ($z_k/z_{\max} \geq 0.8$), then it belongs to the set of dominant frequency candidates \mathcal{D}_f :

$$\mathcal{D}_f = \{f_k \mid z_k \geq 3 \text{ and } z_k/z_{\max} \geq 0.8\} \quad (3)$$

Depending on the number of candidates, we distinguish several cases. If $\mathcal{D}_f = \{f_k\}$ (single candidate frequency), we have a high confidence that the signal is periodic with the dominant frequency $f_d = f_k$. If $\mathcal{D}_f = \{f_{k_1}, f_{k_2}\}$ (two candidate frequencies), the signal has some variation in its behavior but is still periodic. In this case, FTIO returns that the dominant frequency is the one with the highest power contribution. Finally, if none or more than two candidates were found, there is no dominant frequency, and the signal is most likely not periodic. There is an exception when the candidates are multiples of two of each other. In this case, the higher frequencies are ignored. The presence of this kind of harmonics with decreasing high contributions is an indication that there are periodic I/O bursts in the signal.

We favored simple calculations in our approach, as we aimed for an implementation with minimal overhead. Aside from the Z-score, FTIO supports other outlier detection methods, including DBSCAN, isolation forest, local outlier factor, and the `find_peaks` algorithm from SciPy that can all deliver decision functions to find the outliers. A key advantage is that several parameters these algorithms require can be easily found. For DBSCAN, for example, the frequency step can be used to compute `eps` (minimal distance between two points to be considered as neighbors). Still, while these algorithms can improve the results (either alone or by merging their result with the Z-score), they often require more computational effort. Thus, the decision to use them depends on the intended use of FTIO. In the next section, we provide metrics that express the confidence in the results of the period extraction.

C. Confidence Metrics

For a better interpretation of the results, FTIO provides the confidence c_k in the frequency f_k in case at most two frequencies are in \mathcal{D}_f . If we call $\mathcal{I}_1 = \{i \mid z_i \geq 3\}$ the set of frequencies that are outliers, and $\mathcal{I}_2 = \{i \mid z_i/z_{\max} \geq 0.8\}$ the set of frequencies whose Z-score is within 80% of the maximum Z-score, then:

$$c_k = \frac{1}{2} \left(\frac{z_k}{\sum_{i \in \mathcal{I}_1} z_i} + \frac{z_k}{\sum_{i \in \mathcal{I}_2} z_i} \right)$$

Consequently, c_d is the confidence of the dominant frequency f_d . To refine this confidence metric, we optionally provide a second method that does not rely on the result from DFT, namely autocorrelation.

Autocorrelation: Another signal analysis method for finding the period is autocorrelation [25]. The autocorrelation function (ACF) measures the correlation of the observations within a time series at various lags [26]. This allows for spotting repeated patterns in a signal. The ACF can attain values in $[-1, 1]$. We compute the ACF using NumPy’s `correlate` function on the discretized signal with all N samples. To find the periods in the signal, we detect the peak locations in the ACF, find the number of samples *between* them, and divide the obtained values by f_s . Unlike DFT, these candidates can be repeated several times. Hence, after filtering outliers (e.g., with Z-score), we find the period of the signal using the

average. Using the coefficient of variation (σ/\bar{p}) we provide a confidence ($c_a = 1 - \sigma/\bar{p}$) in our result from autocorrelation.

As the period from the autocorrelation is found using averaging, we trust the DFT result more. Consequently, if the results are merged, the autocorrelation results are used to adjust the confidence obtained from DFT. For that, we find the similarity of the dominant frequency from DFT to the candidates from the autocorrelation using the coefficient of variation c_s . Finally, the *refined confidence* is computed by averaging $(c_d + c_a + c_s)/3$. Thus, the *refined confidence* is more reliable as different methods found a similar solution.

Practical example: We executed the IOR benchmark with 9216 ranks on Lichtenberg cluster (described in Section III-B). We set up IOR with 8 iterations, 2 segments, a transfer size of 2 MB, and a block size of 10 MB with the MPI-IO API in the parallel mode and our library preloaded. After the execution on the cluster, we run FTIO on the result for the entire time window Δt of 781 s (i.e., from 64.97 s to 846.7 s) with a sampling frequency $f_s = 10$ Hz. This resulted in 7817 samples and an abstraction error (difference between discrete and original signal) of 0.03. As we only examine half the power spectrum, the number of inspected frequencies is 3809 and the maximum value on the x-axis of the spectrum is 5 Hz. FTIO detected that the signal has a period of 111.67 s (i.e., 0.01 Hz) as the top part of Figure 2 (cosine wave frequency) shows, with a confidence of $c_d = 60.5\%$. The lower part of Figure 2 shows the normed power spectrum zoomed to the relevant frequencies. On average, each of the 3809 frequencies contributed 0.025% to the power. As shown, the frequency at 0.01 Hz has the highest contribution. If the tolerance value is lowered from 0.8 to 0.45, the frequency at 0.02 Hz becomes a candidate as well. However, it is a harmonic (multiple of 2) of the dominant frequency and hence is ignored. This increases the confidence to $c_d = 62.5\%$, as the number of candidates

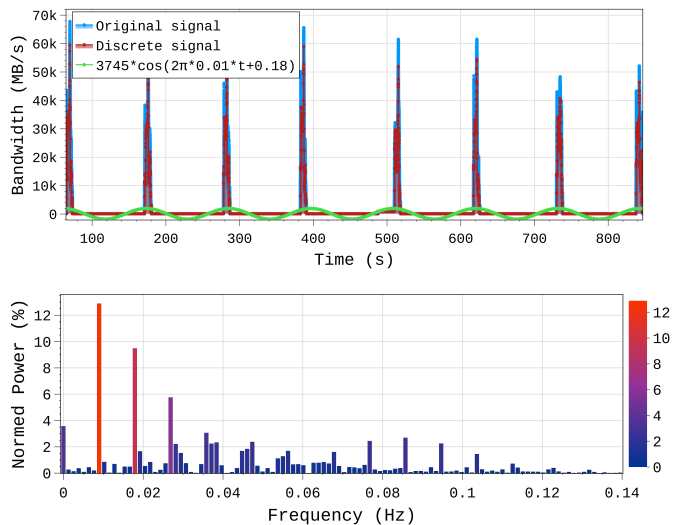


Fig. 2: FTIO results on IOR with 9216 ranks executed on the Lichtenberg cluster. The time behavior (top) and the normed power spectrum (bottom) are shown.

that satisfy $z_k \geq 3$ decreases. As observed in the figure, the presence of such candidates for the dominant frequency indicates the presence of periodic I/O *bursts* in the signal.

In Figure 3, the ACF is plotted against the lag measured in samples for the same signal. Clearly, the correlation of the signal with itself at zero lag is one. Using the `find_peaks` algorithm from SciPy (with a threshold of 0.15), we detected the peaks in the ACF (marked as green triangles in the figure). Next, we divide the samples between consecutive peaks by f_s

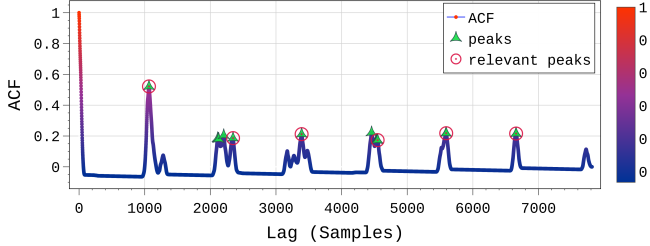


Fig. 3: Result of autocorrelation on IOR with 9216 ranks.

to obtain 17 periods. Using the Z-score with the *weighted mean* (weights from the ACF), we filter the 12 outliers and thus find 5 candidates. Finally, we compute the average with the candidates to obtain a period of 104.8 s (i.e., 0.01 Hz). Note, that these candidates are found using the number of samples *between* the peaks which are marked with red circles in Figure 3. Using the coefficient of variation, we obtain a confidence $c_a = 99.58\%$ in the result from autocorrelation. Finally, we compute the similarity of f_d from DFT to the 5 candidates from the autocorrelation ($c_s = 97.6\%$), and average the three values (62.5%, 99.58%, and 97.6%) to obtain a refined confidence of 86.5%. Thus, by additionally using autocorrelation, we can refine the confidence in the results.

Further characterization: Often, we want to know how much the I/O phases match the result of FTIO or how far from being periodic a signal is, or how long an I/O phase is inside a period. In the rest of this section, we provide metrics to support these aspects and thus allow for additional characterization of the result from FTIO.

a) Standard deviation of volume (σ_{vol}): If we assume an application is periodic, and we know its frequency, then in every period roughly the same amount of data is transferred. For an I/O trace \mathcal{T} , let $V(\mathcal{T})$ be the amount of data accessed in it (i.e., the *volume* of I/O). Given f_d from FTIO, we divide the trace into sub-traces $\{\mathcal{T}_1, \dots, \mathcal{T}_m\}$ each of length $1/f_d$ and data volume $V(\mathcal{T}_i)$ for $i \leq m$. We compute σ_{vol} as the standard deviation of $\frac{V(\mathcal{T}_i)}{\max(V(\mathcal{T}_i))}$. The lower this value, the more similar the data volumes accessed per period are, and thus the more periodic the signal.

b) Time ratio spent on substantial I/O (R_{IO}): An application could be periodic (in time) but not accessing the same amount of data per I/O phase (i.e., high σ_{vol}). We define σ_{time} to evaluate the (time) periodic behavior. Before that, however, we first need to define the time ratio spent on substantial I/O.

Consider, for example, an application that has frequent low-bandwidth I/O, constantly writing a small log file, interleaved

with periodic higher-bandwidth I/O phases. In this case, we consider the low-bandwidth activity as *noise* and the I/O phases as *substantial I/O*. On the contrary, for a signal composed only of the same low-bandwidth “noise,” we might not want to consider it as noise but as the I/O behavior of the application. Therefore, we need a threshold of what is noise and what is not for each application. A fixed per-system threshold could be enough for some usage of this method (e.g., I/O scheduling), but here, we focus on the more challenging and generic case. For the trace \mathcal{T} of length $L(\mathcal{T})$, we set the threshold as $V(\mathcal{T})/L(\mathcal{T})$. Let S be the subset of the trace where the volume of I/O per time-unit is greater than this threshold. Having filtered out the noise, we can compute the time ratio spent doing substantial I/O:

$$R_{IO} = L(S)/L(\mathcal{T}),$$

with $R_{IO} \in [0, 1]$. We can also identify the bandwidth characterizing the *substantial I/O* of the whole trace:

$$B_{IO} = V(S)/L(S)$$

This is illustrated in Figure 4. Moreover, the amount of data transferred per period can be easily calculated by $\frac{V(S)}{L(\mathcal{T}) \times f_d}$. The lower σ_{vol} , the better this value works as a prediction for a future I/O phase.

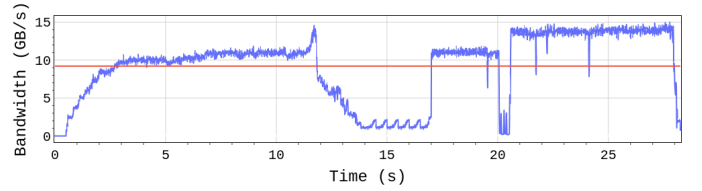


Fig. 4: A red line marks the $V(\mathcal{T})/L(\mathcal{T})$ threshold in the trace from Figure 1. Here $R_{IO} = 0.68$ and $B_{IO} \approx 11$ GB/s.

c) Standard deviation of time (σ_{time}): Similarly to S , let S_i be the subset of \mathcal{T}_i where the volume of I/O per time-unit is greater than $V(\mathcal{T})/L(\mathcal{T})$. Then σ_{time} is defined as:

$$\sigma_{time} = \sqrt{\frac{1}{L(\mathcal{T}) \cdot f_d} \sum_{i=1}^{L(\mathcal{T}) \cdot f_d} \left(\frac{L(S_i)}{L(\mathcal{T}_i)} - R_{IO} \right)^2} \quad (4)$$

Thus, σ_{time} is the standard deviation of the proportion of time spent on I/O *inside each period*. The intuition is that, if the signal is periodic and the application spends, e.g., 60% of its time on I/O ($R_{IO} = 0.6$), then each of its I/O phases will last approximately 60% of a period. Therefore, the lower the σ_{time} , the more periodic the signal is expected to be.

Values close to zero for both σ_{time} and σ_{vol} indicate a signal that is periodic and, therefore, additionally increase our confidence in the period obtained with FTIO. On the other hand, a high σ_{vol} with a low σ_{time} indicates the application is probably periodic but does not access similar amounts of data per I/O phase. Since both σ_{vol} and σ_{time} are in $[0, 0.5]$, we can provide a *periodicity score* (in $[0, 1]$) for the signal according to the FTIO-provided period as $1 - \sigma_{vol} - \sigma_{time}$.

D. Online Period Prediction

So far, we described the offline (post-mortem) detection approach. For online prediction (during application execution), the approach is similar, with the difference that FTIO is executed, to find f_d and c_d (if any), in a new child process every time new I/O measurements are appended to the trace file. Figure 5 shows an overview of the online methodology. To

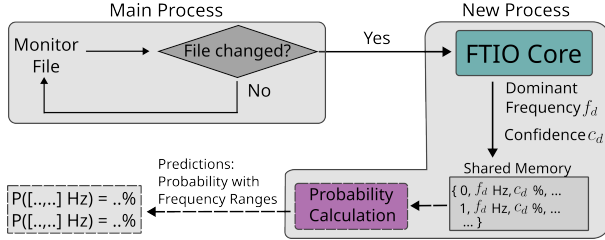


Fig. 5: Online period prediction using FTIO.

adapt to changing I/O behavior and variability, our algorithm offers two optional enhancements: (1) adapting time windows and (2) probability calculations with frequency intervals.

As the I/O behavior of an application can change, it makes sense to discard the old data at some point, and hence, consider a shorter time window for the analysis. Different strategies can be used here. The simplest one is that after finding k times a dominant frequency, the time window for evaluation is reduced to k times the last found period. Alternatively, one could specify a fixed length or a fixed k . Time window adaptation is demonstrated later in Section III-B.

The second enhancement uses the results from consecutive FTIO evaluations, which are stored in a shared memory between the processes. As different executions usually have different time windows, the resolution in the frequency domain changes (see Section II-B). Consequently, when merging predictions, intervals are used. For that, our approach merges the dominant frequencies using DBSCAN with eps set to the difference between the time windows. For each cluster, an interval is calculated by finding the minimum and maximum of the dominant frequencies contained in the cluster. Moreover, the number of predictions inside a cluster divided by the total number of predictions represents the probability of the interval.

E. Parameter Selection

Three parameters affect the analysis: the time window Δt , the sampling frequency f_s , and the number of samples $N = \Delta t \cdot f_s$. The granularity at which the data is captured is specified with f_s . As our approach captures the time spent on each I/O request, we can find the smallest change in bandwidth over time and use it to calculate f_s . However, this is often unnecessary, as we are usually not interested in high-frequency behavior. In contrast, a too-low sampling frequency could result in aliasing. The importance of this is illustrated in Figure 6, which shows the results of FTIO on miniIO [27] executed with 144 ranks on the Lichtenberg cluster. The *unstruct* mini-app was used, which produces unstructured grids with 1000 points per task. In Figure 6, we set f_s to 100 Hz,

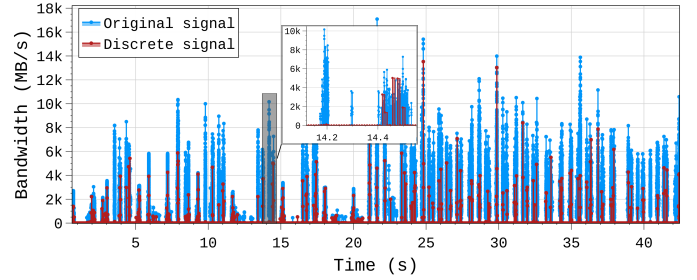


Fig. 6: miniIO with 144 ranks on the Lichtenberg cluster.

which is not enough: the discrete signal does not match the original one at all. But even if the approach had found a single period, the result cannot be trusted, as the abstraction error (the volume difference between the two shown signals) is just too large.

With a constant sampling frequency f_s , increasing Δt increases the number of samples $N = \Delta t \cdot f_s$, which increases the detection/prediction time. In all of our experiments, this time was negligible. Moreover, it does *not* represent overhead to applications, since the analysis is *not* done on the nodes where they run. The only overhead there comes from the tracing library and is analyzed in Section III-C.

III. EVALUATION

In this section, we evaluate FTIO by: (1) analyzing its accuracy and limitations (Section III-A), (2) demonstrating it based on three case studies (Section III-B), and (3) examining its overhead (Section III-C).

A. Limitations of FTIO

In what follows, we explore the accuracy and limitations of FTIO by crafting challenging synthetic traces.

Methodology: We have created “semi-synthetic” traces to allow for an extensive evaluation: First, we traced IOR [28] runs that represent a single I/O phase. Then, we generated application traces by combining I/O phases with a given amount of “idle” (no I/O) time between them. IOR was executed 100 times on the PlaFRIM cluster using 32 processes on four nodes. Each of them writes a 3.5 GB file in 1 MB contiguous requests. One I/O phase was filtered out for being too long compared to the others (due to contention in the system), leaving 99 traces with an average duration of 10.4 s (≈ 10 GB/s), all inside [10.22, 13.34] s.

An application is considered to be a sequence of J non-overlapping iterations. Each iteration $j \leq J$ has a compute phase of length $t_{cpu}^{(j)}$ followed by an I/O phase (of length $t_{io}^{(j)}$) where each of the P processes writes an amount of data v to the file system. The trace is created by selecting J and P , and then, for each $j \leq J$, by:

- Drawing $t_{cpu}^{(j)}$ from a normal distribution $\mathcal{N}(\mu, \sigma)$ truncated to only select positive values (with μ and σ denoting the mean and standard deviation of t_{cpu} , respectively);
- Randomly picking one of the I/O phase traces, which consists of P per-process traces;

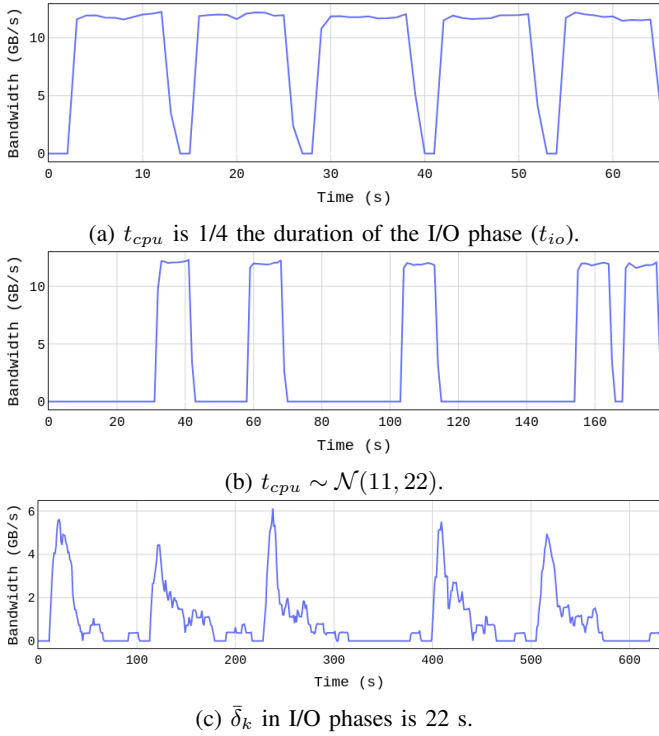


Fig. 7: Examples of “semi-synthetic” application traces.

- For each process $k \in [1, P]$, adding a time δ_k at the beginning of its trace (without I/O). δ_k is drawn from an exponential distribution of average ϕ . Process 0 has $\delta_0 = 0$ to keep the boundaries of the I/O phase.

As the I/O phases’ length depends on δ_k , it allows us to represent both desynchronization between processes and I/O variability. Finally, for experiments with noise, we generated 200 traces from IOR on a single process in two configurations: low noise of nearly 500 MB/s and high noise of nearly 1 GB/s. The noise traces have 10 periods of approximately 2.2 s each. Noise is emulated by randomly selecting a sequence of noise traces and adding them to the application trace.

For all experiments in this part, we used $f_s = 1$ Hz, $P = 32$ (the number of processes used for IOR), and $J = 20$ (to be able to induce enough variability in each trace). Figure 7 illustrates traces created with this approach. We generate 100 traces per parameter combination. For each one, we compute \bar{T} the average period length and T_d the period obtained with FTIO. Finally, we calculate the *detection error* as $|T_d - \bar{T}|/\bar{T}$. Note that \bar{T} can only be computed using information from the trace generation, as the boundaries of I/O phases are not typically available. Reaching a low error means FTIO provides a value that is close to the average period.

Results: First, we study the impact of length differences between CPU and I/O phases, e.g., as seen in Figure 2 and later in Figure 10. For this, we use traces with $\delta_k = 0$ and vary t_{cpu} (with $\sigma = 0$). Figure 8a presents the results and shows that the disparity in phase duration is *not* a problem. They also seem to indicate that when the time between I/O phases

is *longer*, our approach leads to better results. However, that might be an artifact of the fixed sampling frequency. Still, all errors are below 1%. These results also suggest that FTIO is fairly robust to noise.

Next, we cover two challenging scenarios at once: (1) when the processes performing the I/O phase are not synchronized (absence of implicit/explicit barriers), and (2) I/O performance variability, with some I/O phases of the application taking longer than others, which is usually the case when accessing a shared file system. For that, we set $t_{cpu} = 11$ s and increase ϕ (the average δ_k). The results are shown in Figure 8b. When ϕ becomes larger than the original duration of I/O phases, there are often periods without I/O activity inside the I/O phases, which makes their detection more difficult. In extreme cases, the error goes up to 100%, but is in general low: Mean of up to 11%, median up to 11%, and third quartile up to 17%.

Finally, we study the case where the time between I/O phases varies during the execution, as in Figures 2 and 10. We control that by drawing t_{cpu} from $\mathcal{N}(\mu, \sigma)$ with $\mu = 11$ s and increasing σ . For this experiment, we use $\delta_k = 0$ and no noise. Note that the use of real I/O traces for the phases introduces natural variability. In Figure 8c, we can see the results vary in quality as the signal becomes less periodic. This figure was zoomed-in to allow the visualization of the box plots. 26 outliers with errors of more than 200% are not shown (out of 3400 traces). They are: 0.4% of the traces with $\mu \leq \sigma < 2\mu$, and 1.9% of the traces with $\sigma \geq 2\mu$. With $0.5\mu \leq \sigma < \mu$, 16% of the traces obtained confidence below 60%, and that number increases to 27% when $\sigma/\mu \geq 1$. The median confidence drops from 96% when $\sigma/\mu < 0.55$ to 63% when $\sigma/\mu \geq 2$. In all cases, the median error remains below 33% (and below 5.5% for $\sigma/\mu \leq 0.5$). For all cases, the calculated R_{IO} is wrong by less than 10%.

Figure 9 presents the metrics σ_{vol} (left) and σ_{time} (right) for this experiment. As shown, both increase as the I/O variability increases (i.e., less periodic signal). Their variability for each point from the x-axis matches the variability observed in the error shown in Figure 8c. The median periodicity score (see Sec. II-C) is 98% for $\sigma = 0$, then drops to 67% for $\sigma/\mu = 0.55$, and to 57% for $\sigma/\mu = 2$. Hence, when designing a technique (e.g., I/O scheduling) that uses the period obtained with FTIO, one can study the robustness of their technique according to the values of σ_{vol} and σ_{time} to decide on thresholds for these metrics, since some approaches will tolerate higher detection errors than others.

B. Case Studies

In Section II-C, we showed the scalability of FTIO using IOR [28]. To evaluate our method further, in this section we: (a) analyze a real application (LAMMPS [29]) with low I/O bandwidth, (b) demonstrate the compatibility of FTIO with a Darshan profile of Nek5000 [30], and (c) use a mini-app (HACC-IO [31]) with high I/O bandwidth to highlight the detection and prediction capabilities of FTIO. The experiments (a) and (c) were performed on the Lichtenberg cluster, where a typical node has 96 cores, and the access mode is user-

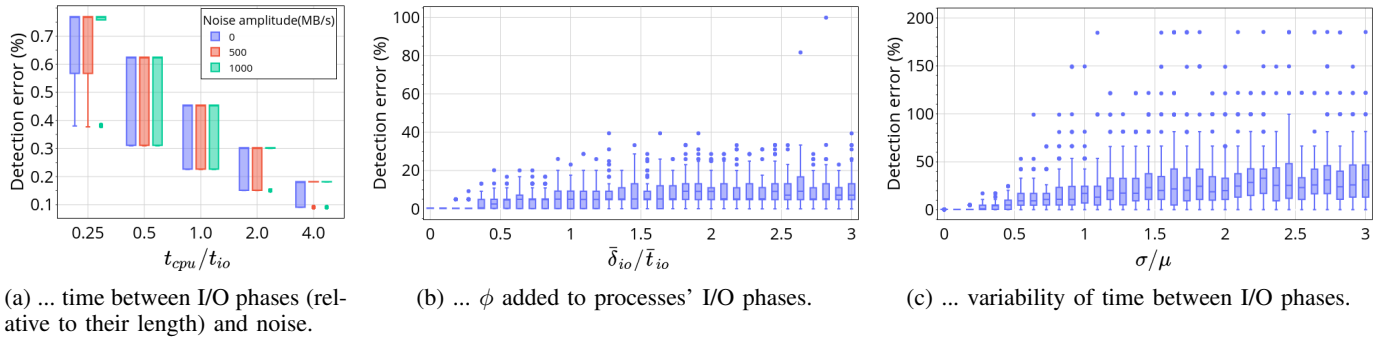


Fig. 8: Detection error as a function of the ...

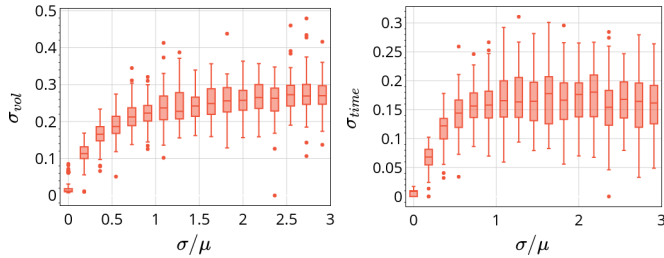


Fig. 9: σ_{vol} and σ_{time} from the experiments in Figure 8c. The term σ/μ on the x-axes represents the standard deviation of t_{cpu} divided by the mean of t_{cpu} .

exclusive. The shared file system (IBM Spectrum Scale) has a peak performance of 106 GB/s for writes and 120 GB/s for reads.

a) Real application with low I/O bandwidth: We demonstrate our approach on LAMMPS [29] with 3072 ranks. We use the 2-d LJ flow simulation with 300 runs that dumps all atoms every 20 runs. Using FTIO with $f_s = 10$ Hz in the detection (offline) mode, the result was obtained in 2.2 s. The top part of Figure 10 shows the single-sided power spectrum. As illustrated, FTIO found a single dominant frequency at 0.039 Hz (25.73 s) with a confidence of 55.0%. Due to the moderate contribution of the frequency at 0.16 Hz (5.9 s), the low confidence is justified. Using autocorrelation (for an additional cost of 0.26 s), the confidence is refined and we obtain 84.9% (with only a single peak detected at 25.6 s). For comparison, the real mean period for this execution was 27.38 s. The bottom part of Figure 10 demonstrates the result of FTIO in the time domain, which shows the low I/O performance due to the writing method. As observed, the dominant frequency does not perfectly fit all phases (e.g., at 143 s), justifying again the low confidence obtained. Still, it provides an adequate and concise representation of the temporal I/O behavior of the application, which is what we aimed for. Note that the results can be improved by adapting Δt , as the next example shows. Still, the offline approach demonstrated here could be used, for example, to feed an I/O scheduler at the start with the period of the I/O phases from previous executions.

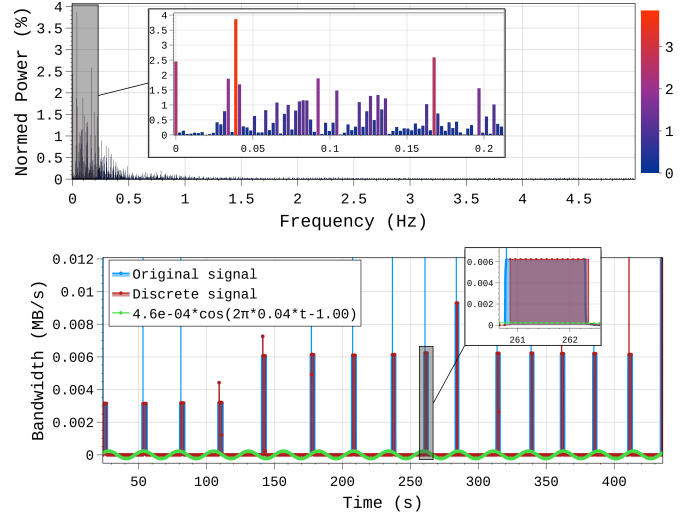


Fig. 10: Result of FTIO on LAMMPS with 3072 ranks: normed power spectrum (top) and time behavior (bottom).

b) Compatibility with other tools: For this example, we downloaded from the *I/O Trace Initiative* website [32] a Darshan profile³ of Nek5000 [30] (turbulence simulation) executed with 2048 ranks on the Mogon II cluster. FTIO extracted the heatmap from Darshan profile and automatically set the sampling frequency to the bin widths in seconds ($f_s = 0.006$ Hz). While a higher sampling frequency could be used here, there is no advantage due to the constant behavior in the bins. FTIO detected that the I/O phases are not periodic if the entire trace is considered ($\Delta t = 86,000$ s) as shown in the lower part of Figure 11. This is due to the irregular I/O phases at roughly 57,000 and 85,000 s, which write each around 30 GB, compared to the 7 GB of the remaining ones (except the phases at 0 s and 45,000 s, which write 13 and 75 GB, respectively). Moreover, the bins that write 7 GB are not equally spaced. However, if the time window is set to $\Delta t = 56,000$ s, FTIO detects a period of 4642.1 s with a confidence of 85.4% as the upper left part of Figure 11 shows. Moreover, the power spectrum is less noisy compared to the previous case shown on the right side of Figure 11, where

³<https://hpcioanalysis.zdv.uni-mainz.de/trace/64ed13e0f9a07cf8244e45cc>

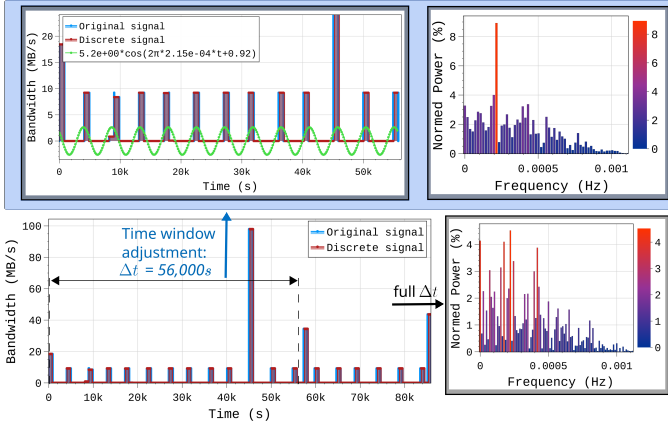


Fig. 11: Result of FTIO on Nek5000 with the full trace (bottom), and a reduced time window $\Delta t = 56,000$ s (top).

we zoomed to the relevant frequencies. Consequently, a clear outlier can be detected. In the next example, we demonstrate how the online prediction automatically adapts Δt to improve the prediction results.

c) Detection and prediction with high I/O bandwidth:

Next, we use HACC-IO [31], which mimics one I/O phase of HACC (Hybrid/Hardware Accelerated Cosmology Code) [33]. HACC-IO has four steps: compute, write, read, and verify. We added a loop around these steps to execute them periodically. Moreover, at the end of each loop iteration, we added a single line to flush the collected data out to the trace file. On a login node, we deployed FTIO in the online prediction mode. We executed this example with 3072 ranks on the Lichtenberg cluster.

1) *Offline evaluation:* We first look into the output of the offline evaluation performed over the whole trace after the end of the execution. Figure 12 presents the single-sided normed power spectrum. Two candidates for the dominant frequency

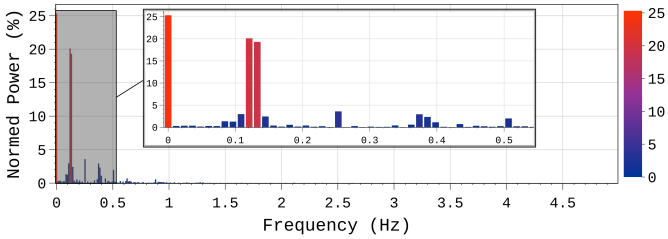


Fig. 12: Single-sided power spectrum from DFT on HACC-IO with 3072 ranks and a sampling frequency $f_s = 10$ Hz.

were found: 0.1206 Hz ($c_k = 51\%$) and 0.1326 Hz ($c_k = 48.9\%$). As the former one has the highest contribution, it is the dominant frequency corresponding to a period of 8.29 s. Note that the application is by design periodic. However, if we study qualitatively the execution in Figure 13, we see that the first I/O phase was significantly delayed: it lasts from 4.1 s to 15.3 s. This changing behavior results in a less periodic signal and explains our moderate confidence. Indeed, the average period is 8.7 s, which becomes 7.7 s without the first I/O

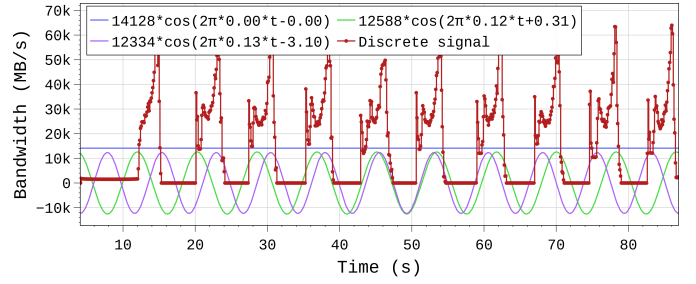


Fig. 13: DC Offset and the highest-contributing frequencies found in the temporal behavior of HACC-IO with 3072 ranks executed on the Lichtenberg cluster.

phase. Figure 13 shows the top three signals found by DFT. As presented, the I/O phases align more with the 0.120 Hz signal (green) at the start and with the 0.132 Hz signal (purple) near the end.

As the two candidates for the dominant frequency have very close contributions and are consecutive, one approach could be to merge them by taking the sum of their cosine waves. This is shown in Figure 14, and would provide a more accurate representation of the application’s temporal I/O behavior. However, in this paper, we focus on representing the

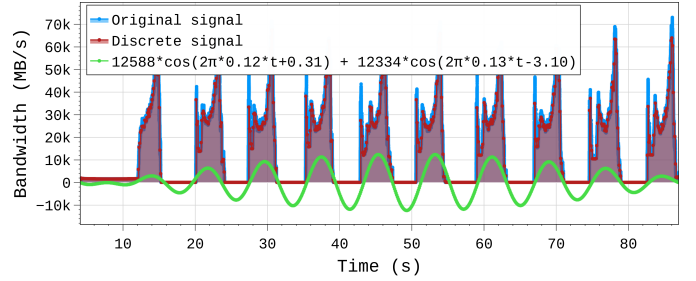
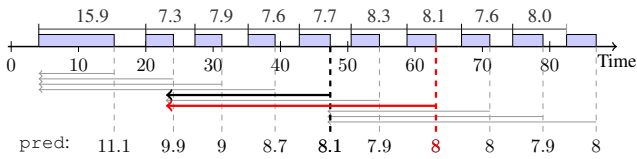


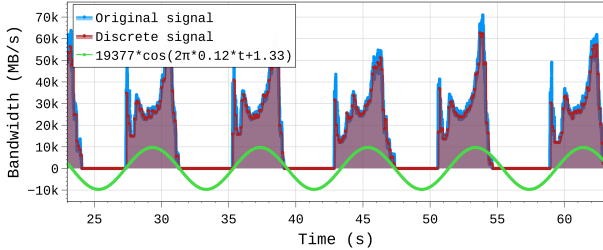
Fig. 14: HACC-IO with 3072 ranks. By combining the dominant frequency candidates, a more accurate description of the application’s temporal I/O behavior can be obtained. However, this is more difficult to interpret, and hence, not used here.

behavior with a single period, which is concise and can easily be used as an input for techniques such as I/O scheduling. In contrast, a more detailed application profile could include several dominant frequency candidates and their contributions. We plan on exploring such profiles in the future. Note that as the first phase is often prolonged due to initialization overheads, FTIO provides an option to skip it. Next, we show how the online version of FTIO automatically handles changing I/O behavior.

2) *Online Prediction:* As discussed in Section II-E, the time window for FTIO predictions can be automatically adapted according to the found frequency. For the ten I/O phases, which started on average every 8.7 s, the average obtained period is 8.66 s. All predictions are shown visually in Figure 15a. As shown, predictions were done at the end of each I/O phase (dashed vertical lines) when new data became available. At the end of the 3rd prediction, a dominant frequency was identified



(a) Above the time axis is the ground truth: the blue rectangles are the I/O phases, and above them is the time between their start times. Below the time axis are the predictions: at time 63 s, we predicted a period of 8 s based on the history up to time 23 s (thick red line).



(b) Details of the 7th prediction (thick red line from above).

Fig. 15: Online prediction during the execution of HACC-IO with 3072 ranks.

for the third time. As the 4th prediction had already started, that information was made available for the 5th prediction. Hence, for the next evaluation, we only kept the data between 23.1 s ($47.4 - 3 \times 8.1 = 23.1$ s) and 47.4 s (time of the 5th prediction). In the figure, this is represented in bold. Similarly, at 8th prediction, the time window was again adapted ($71.1 - 3 \times 8 = 47.3$ s).

In this section we demonstrated the use of FTIO with large-scale applications for both offline detection and online prediction alongside metrics that gauge the confidence in our results. I/O variability and changing behaviors often caused the signal to be less periodic, resulting in a moderate confidence in the obtained results. The observation from HACC-IO indicates that, in these situations, the online prediction approach can yield the best results by adapting the time window.

C. Overhead of the Tracing Library

The tracing library can be used for offline detection or online prediction. From those, we examine the online approach as it has a higher overhead since it sends information to the file more often (see Section II-A). To measure it, we executed IOR with the same settings as in Section II-B, on the Lichtenberg cluster (see Section III-B) with different numbers of processes (all multiples of 96 as we have 96 cores per typical node).

Figure 16 shows the overhead of the tracing library, with the top part showing the aggregated time, while the bottom plot shows the time from the MPI rank 0 perspective. The numbers of ranks on the x-axis are in log scale, and the sum of the application time (App) and overhead is the total time. More precisely, we instrumented our library calls and subtracted the corresponding values from the measured total time to derive the application time. As observed, for capturing and logging the I/O data, our tracing library has a low overhead:

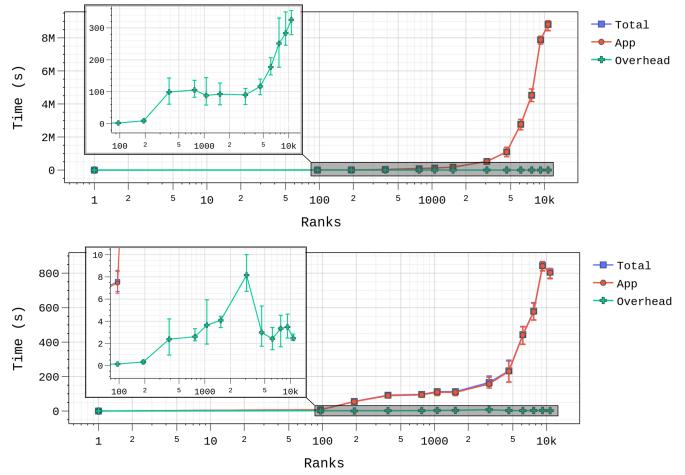


Fig. 16: Overhead of our approach across different rank configurations (from 1 till 10752 ranks). The top shows the aggregated time across all ranks, while the bottom shows the time of MPI rank 0 only.

a maximum of 0.6% for the aggregated overhead and 6.9% for the overhead for rank 0 only. The data gathering from the different ranks is the major source of overhead. For comparison, in the same configurations, the overhead of the offline approach ranged from 0.78 s (0.13%) at 96 ranks to 50.9 s (0.004%) at 4608 ranks in the aggregated overhead time and increased nearly linearly from 0.065 s (1.03%) to 3.84 s (1.58%) for the overhead for rank 0 only.

It is important to notice that our approach can be used with other data collection strategies (see Section II-A) which would have different implications as demonstrated in the second example in Section III-B. The execution time of the analysis is of minor importance, as mentioned, and depends on the length of the time window. For all examples in this paper, the longest analyses took: 2.2 s for LAMMPS, 5.7 s (5.9 s with autocorrelation) for IOR, 8.7 s (8.5 s with adjusted Δt) for Nek5000 of which pyDarshan consumed 3.8 s to import the data, and 3.6 s for the offline detection for HACC-IO.

IV. USE CASE: I/O SCHEDULING

Although work on I/O scheduling [7], [9], [10], [21] has proved how critical the knowledge of I/O phases is, FTIO is the first runtime solution that provides simple and lightweight access to this information. To illustrate the utility of FTIO, we coupled it with Set-10 [10], an I/O scheduling heuristic. The goal of Set-10 is to mitigate file-system contention, exploiting that the frequencies at which jobs perform their I/O usually differ. For this purpose, Set-10 groups jobs according to their I/O period. It then grants shared file-system access to different groups (based on priorities) and mutually exclusive access to individual jobs within the same group. In case FTIO is used together with Set-10 (later denoted as "Set-10 + FTIO"), the priorities for the groups (i.e., the sets) are calculated based on the period T_d provided by FTIO. According to the Set-10 algorithm, applications with the smallest period receive the

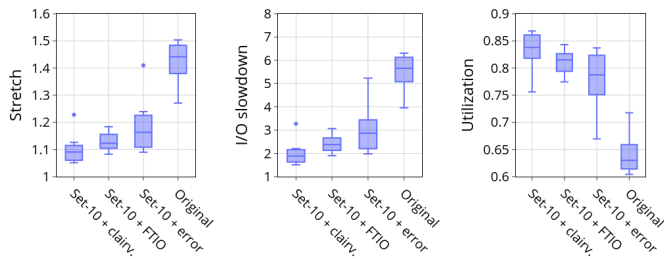


Fig. 17: Comparison of clairvoyant Set-10, Set-10 with FTIO, Set-10 with 50% error injected to the FTIO-provided periods, and the original configuration without Set-10. The figures show the stretch (how much slower jobs were compared to running in isolation: lower is better), the I/O slowdown (how much slower I/O was compared to isolation: lower is better), and the utilization (how much of the time was NOT spent on I/O: higher is better). The boxplots (with 1.5*IQR whiskers) group ten executions. The y-axes do not start at zero and are all different.

highest priority and, therefore, most of the bandwidth [10]. Note that in the original Set-10 implementation, priorities for the groups were calculated using the characteristic time w_{iter} (i.e., the average time between the beginning of two consecutive I/O phases) as described in [10, Section IV-C].

For our experiments, we used an implementation, described in [34], of Set-10 on BeeGFS and deployed FTIO to determine each job’s period at runtime. Our workload consists of one high-frequency and 15 low-frequency applications derived from the IOR benchmark. They were designed to include, in isolation, periods of 19.2 s (high frequency) or 384 s (low frequency), with I/O consuming 6.25% of each period. Figure 17 shows the results, comparing four situations:

- “Set-10 + clairv.” is a clairvoyant application of the scheduling heuristic, meaning that the *ideal* (in isolation) periods (19.2 or 384 s) are provided manually in advance.
- “Set-10 + FTIO” combines the heuristic with FTIO, which determines the *actual* periods at runtime. In this case, Set-10 uses the most recent prediction from FTIO.
- “Set-10 + error” uses predictions *worse* than FTIO: the predictions given by FTIO are randomly increased or decreased by a factor of 50% before provided to Set-10.
- “Original” corresponds to BeeGFS without any modifications and serves as the baseline.

We evaluate these algorithms with three metrics: the stretch, the I/O slowdown, and the system utilization. The *stretch* quantifies the overall slowdown factor for an application caused by inter-job file-system interference; the *I/O slowdown* represents the factor by which its I/O time was increased. Thus, the lowest value of both metrics is 1. Both are calculated by taking the geometric mean of all applications from each execution. The *Utilization* ($\in [0, 1]$) is a system metric that specifies how much of the node time was spent on computation instead of I/O. More details about these metrics are given

in [10, Section V-D], and more on this experiment in [34].

The results achieved with FTIO are close to the clairvoyant version—only 2.2% worse in stretch, 19% in I/O slowdown, and 2.3% in utilization. In contrast, the version where we inject errors to FTIO results made stretch worse by 5%, utilization by 4%, and I/O slowdown 27% higher, compared to the “Set-10 + FTIO” version, in addition to presenting higher variability. Thus, the main observation from Figure 17 is that FTIO hits a sweet spot in performance where a better prediction would not improve the performance observed by the system or the users, however, a worse prediction would increase the variability and impair the performance of the system. That indicates FTIO provides results that are good enough for Set-10, and a more accurate method, if available, would not have much margin to improve the results significantly. Compared to not using Set-10, the FTIO-powered version decreased the mean stretch and I/O slowdown by 20% and 56%, respectively, and increased utilization by 26%. These results show how well FTIO fills the knowledge gap, making the improvements that Set-10 allows possible in practice, where the period is not known in advance.

V. RELATED WORK

Since I/O performance depends on many parameters [6], [20], [35]–[37], profiling tools such as Darshan [1], [18] and other more holistic approaches [38], [39] can be used by an expert to obtain insights about application I/O behavior and improve it. However, these large profiles are not easily automatically exploitable at run time by optimization techniques, which must focus on simpler metrics. For example, in the context of cache management and I/O prefetching, it is useful to predict future I/O requests [40]. That has been done using neural networks [4], ARIMA time series analysis [41], pattern matching [42], context-free grammars [2], etc. Although FTIO could be used to predict future accesses, it is fundamentally different from these approaches because we focus on *I/O phases*, not *I/O requests*. Working at this higher level brings the challenge of not knowing when the I/O phases start and end (see Section I), particularly since the phases are logical groupings of I/O requests, not individual events. Still, the period of the I/O phases is a metric worth finding as it can be easily and directly exploited by contention avoidance algorithm, as demonstrated in Section IV.

Aside from being able to handle changing I/O behavior (see Section II-D), FTIO can be executed online due to its low overhead (see Section III-C). The main advantages of FTIO in this comparison basically stream from the unique properties of DFT. Compared to popular machine learning (ML) approaches from the time domain, like neuronal networks (NN) [43] and LSTMs [44], [45], decision trees and other supervised methods [46], or a combination of supervised and unsupervised techniques [20], FTIO, and in particular DFT, does not require a learning phase. Additionally, FTIO does not require past system logs, different from recent regression-based approaches [47] and other strategies [20], [44]–[48]. Moreover, compared to approaches that predict future I/O activity, such as ARIMA [41], DFT does not require defining

several thresholds and parameter estimations. In contrast, DFT yields the signal’s frequency components rather than providing a detailed time model. Still, this is enough to predict the *period* of the I/O phases, as demonstrated in Section III, and in particular for the I/O scheduling use case (Section IV).

Determining the application-level period from time models usually requires defining thresholds, which can be system and application-dependent (see Section I). Even if an approach, such as ARIMA, accurately predicted the time behavior shown in Figure 1, we would still need to analyze the result further to extract the period. In this context, finding suitable thresholds to detect the phases is challenging, primarily due to the varying nature of I/O (e.g., I/O burst, slow I/O, etc.). FTIO directly utilizes DFT to overcome such challenges and, combined with outlier detection methods, determines the period of the I/O phases. Further characterization can be provided based on the identified period as described at the end of Section II-C. Additionally, as shown in Section III-B, the parameters of DFT allow FTIO to adapt to changing behavior (using Δt) and to specify the range of interesting I/O (using f_s). Furthermore, the online time window adaptation usually decreases the overhead of FTIO further, as less samples (N) are included in the analysis (see Section II-B), making this approach even more favorable for online period prediction.

More general characterization efforts usually focus on aspects such as spatiality and request size [49], [50], using information from MPI-IO [51]–[53], ML-based methods [43], etc. In contrast, FTIO focuses on the *temporal* behavior (specifically on the periodicity), and hence is also complementary to those. Other recent approaches allow to identify the number of phases manually by adjusting a threshold [54], and consequently extract the period visually. In contrast, FTIO not only automatically extracts the period and provides a metric for confidence but also performs this during an application’s runtime in the online mode. Still, a combination of both approaches might be very valuable for the HPC community.

In the field of performance analysis, Casas et al. [55] proposed to construct signals of metrics (e.g., number of active processes, amount of communicated data, etc.), and then to apply discrete wavelet transform to keep the highest-frequency portions and autocorrelation to find the frequency of the application’s phases. We were inspired by the use of signal processing techniques, but our approach is different since we advocate a lightweight approach to concisely represent the period of the I/O behavior, whereas they aim at removing the effects of external noise to detect the phases that best represent the application. Yang et al. [19] introduced a metric to quantify the burstiness of I/O and apply it to traces from a production machine. They found that most traces presented a very high degree of burstiness; however, their metric is a measure of “unevenness”, not of periodicity, which is our focus. Qiao et al [56] used DFT on a signal of write performance over function calls. They used it to search for the period of other concurrent applications (and use that to predict future interference). We argue for a scenario where this information can be easily obtained for all applications and shared so that

smart decisions can be made throughout the system.

VI. CONCLUSION

This paper presents FTIO, an approach for characterizing and predicting the temporal I/O behavior of an application with a simple metric, namely its period, obtained using different frequency techniques. We provided several extensions to adapt to the unsteady nature of I/O and further describe the behavior. Our evaluation demonstrates the low overhead of FTIO, its suitability for real large-scale examples, and its robustness with a mean error below 11%. Combined with the I/O scheduler Set-10, FTIO allowed increasing system utilization by 26% and decreasing I/O slowdown by 56%. However, I/O scheduling is only one possible application of FTIO. Its predictions could also be helpful in other contexts, such as burst buffer management (e.g., flushing before the buffer is full to overcome storage space restrictions). Moreover, the post-mortem analysis could be used for I/O-aware batch scheduling.

Despite our focus on whole applications, there are use cases (e.g., cache management) which require knowing the behavior of individual processes. Even in such cases, our approach is equally suitable. Furthermore, although we focused on I/O in this paper, our technique can be repurposed for other use cases (e.g., finding the period of scheduling points) by simply changing the input data. Finally, our discussion on the selection of the sampling frequency (Section II-E) assumes we are interested in *any* frequency the application’s I/O behavior exhibits. In some cases, such as I/O scheduling, for example, we may *not* be interested in high frequencies because we cannot respond fast enough, so the sampling frequency (f_s) could act as a filter. Future work will focus on exploring online f_s adaptation. Moreover, our approach rests on DFT, which has a high-frequency resolution but no time resolution. We plan to explore merging the result with the wavelet transform [57] for a more comprehensive characterization, to prepare for cases where we need both.

ACKNOWLEDGMENT

The authors would like to thank Jean-Baptiste Besnard (ParaTools) for his support and enthusiasm for this work. The authors also thank Clément Barthélemy and Luan Teylo for their help in setting up the Set-10 experiments.

APPENDIX: ARTIFACTS REPRODUCIBILITY

As mentioned at the beginning of Section II, FTIO and TMIO are publicly available on GitHub. The FTIO repository⁴ provides descriptions on how to reproduce the results and experiments from this paper. Additionally, the data sets from these experiments are publicly available [58].

⁴<https://github.com/tuda-parallel/FTIO/tree/main/artifacts/ipdps24>

REFERENCES

- [1] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale I/O workloads," in *Cluster'09 Workshops*. IEEE, 2009, pp. 1–10.
- [2] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Using Formal Grammars to Predict I/O Behaviors in HPC: the OmniscIO Approach," *IEEE Transactions on Parallel and Distributed Systems*, 2016. [Online]. Available: <https://hal.inria.fr/hal-01238103>
- [3] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC applications under congestion," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 1013–1022.
- [4] S.-M. Tseng, B. Nicolae, G. Bosilca, E. Jeannot, A. Chandramowlishwaran, and F. Cappello, "Towards portable online prediction of network utilization using MPI-level monitoring," in *Euro-Par 2019: Parallel Processing*, R. Yahyapour, Ed. Cham: Springer International Publishing, 2019, pp. 47–60.
- [5] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the root causes of cross-application I/O interference in HPC storage systems," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 750–759.
- [6] T. Wang, S. Byna, G. K. Lockwood, S. Snyder, P. Carns, S. Kim, and N. J. Wright, "A zoom-in analysis of I/O logs to detect root causes of I/O performance bottlenecks," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 102–111.
- [7] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination," in *IPDPS'14*. IEEE, 2014, pp. 155–164.
- [8] Z. Zhou, X. Yang, D. Zhao, P. Rich, W. Tang, J. Wang, and Z. Lan, "I/O-aware batch scheduling for petascale computing systems," in *Cluster'15*. IEEE, 2015, pp. 254–263.
- [9] E. Jeannot, G. Pallez, and N. Vidal, "Scheduling periodic I/O access with bi-colored chains: models and algorithms," *J. of Scheduling*, vol. 24, no. 5, pp. 469–481, 2021.
- [10] F. Boito, G. Pallez, L. Teylo, and N. Vidal, "IO-sets: Simple and efficient approaches for I/O bandwidth management," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 10, pp. 2783–2796, 2023.
- [11] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 455–466, Aug. 2014.
- [12] T. T. Tran, M. Padmanabhan, P. Y. Zhang, H. Li, D. G. Down, and J. C. Beck, "Multi-stage resource-aware scheduling for data centers with heterogeneous servers," *J. of Scheduling*, vol. 21, no. 2, pp. 251–267, Apr. 2018.
- [13] R. Bleuse, K. Dogeas, G. Lucarelli, G. Mounié, and D. Trystram, "Interference-aware scheduling using geometric constraints," in *Euro-Par'18*. Springer, 2018, pp. 205–217.
- [14] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Server-Side Log Data Analytics for I/O Workload Characterization and Coordination on Large Shared Storage Systems," in *SC'16*, Nov 2016, pp. 819–829.
- [15] H. Sung, J. Bang, C. Kim, H.-S. Kim, A. Sim, G. K. Lockwood, and H. Eom, "BBOS: Efficient HPC Storage Management via Burst Buffer Over-Subscription," in *CCGrid'20*, 2020, pp. 142–151.
- [16] G. Aupy, O. Beaumont, and L. Eyraud-Dubois, "What size should your buffers to disks be?" in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 660–669.
- [17] —, "Sizing and partitioning strategies for burst-buffers to reduce IO contention," in *IPDPS'19*. IEEE, 2019, pp. 631–640.
- [18] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular HPC I/O characterization with Darshan," in *2016 5th workshop on extreme-scale programming tools (ESPT)*. IEEE, 2016, pp. 9–17.
- [19] W. Yang, X. Liao, D. Dong, and J. Yu, "A quantitative study of the spatiotemporal I/O burstiness of HPC application," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 1349–1359.
- [20] M. Isakov, E. d. Rosario, S. Madireddy, P. Balaprakash, P. Carns, R. B. Ross, and M. A. Kinsy, "HPC I/O Throughput Bottleneck Analysis with Explainable Local Models," in *SC'20*, 2020, pp. 1–13.
- [21] A. Benoit, T. Herault, L. Perotin, Y. Robert, and F. Vivien, "Revisiting I/O bandwidth-sharing strategies for HPC applications," INRIA, Tech. Rep. RR-9502, Mar. 2023. [Online]. Available: <https://hal.inria.fr/hal-04038011>
- [22] "Messagepack: It's like json. but fast and small." <https://msgpack.org/index.html>, Sep 2023. [Online]. Available: <https://github.com/msgpack/msgpack>
- [23] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient Parallel I/O Tracing and Analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2020, pp. 1–8.
- [24] K. S. Kannan, K. Manoj, and S. Arumugam, "Labeling methods for identifying outliers," *International Journal of Statistics and Systems*, vol. 10, no. 2, pp. 231–238, 2015.
- [25] M. N. Nounou and B. R. Bakshi, *Chapter 5 - multiscale methods for denoising and compression*, ser. Data handling in science and technology. Elsevier, 2000, vol. 22, p. 119–150. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0922348700800301>
- [26] G. Box and G. Jenkins, *Time series analysis: Forecasting and control*, ser. Holden-Day series in time series analysis and digital processing. Holden-Day, 1976. [Online]. Available: <https://books.google.de/books?id=1WVHAAAAAAAJ>
- [27] miniIO Benchmark, <https://github.com/PETTT/miniIO>, 2022.
- [28] I. Benchmark, "Version 3.3.0," <https://github.com/hpc/ior>, 2020.
- [29] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Comp. Phys. Comm.*, vol. 271, p. 108171, 2022.
- [30] J. W. L. Paul F. Fischer and S. G. Kerkemeier, "NEK5000 web page," 2008, <http://nek5000.mcs.anl.gov>.
- [31] LLNL, "CORAL Benchmark Codes - HACC IO," <https://asc.llnl.gov/coral-benchmarks#hacc>, 2020.
- [32] N. Moti, A. Brinkmann, M. Vef, P. Deniel, J. Carretero, P. H. Carns, J. Acquaviva, and R. Salkhordeh, "The I/O trace initiative: Building a collaborative I/O archive to advance HPC," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W 2023, Denver, CO, USA, November 12-17, 2023*. ACM, 2023, pp. 1216–1222.
- [33] S. Habib, V. Morozov, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, T. Peterka, J. Insley, D. Daniel, P. Fasel, N. Frontiere, and Z. Lucic, "The Universe at extreme scale: Multi-petaflop sky simulation on the BG/Q," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. Salt Lake City, UT: IEEE, Nov. 2012, pp. 1–11.
- [34] C. Barthélemy, F. Boito, E. Jeannot, G. Pallez, and L. Teylo, "Implementation of an unbalanced I/O bandwidth management system in a parallel file system," 2024, available at <https://inria.hal.science/hal-04417412>.
- [35] L. Pottier, R. F. da Silva, H. Casanova, and E. Deelman, "Modeling the performance of scientific workflow executions on HPC platforms with burst buffers," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 92–103.
- [36] F. Chowdhury, Y. Zhu, T. Heer, S. Paredes, A. Moody, R. Goldstone, K. Mohror, and W. Yu, "I/O characterization and performance evaluation of beegfs for deep learning," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: <https://doi.org/10.1145/3337821.3337902>
- [37] B. Yang, X. Ji, X. Ma, X. Wang, T. Zhang, X. Zhu, N. El-Sayed, H. Lan, Y. Yang, J. Zhai, W. Liu, and W. Xue, "End-to-end I/O monitoring on a leading supercomputer," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'19. USA: USENIX Association, 2019, p. 379–394.
- [38] G. K. Lockwood, W. Yoo, S. Byna, N. J. Wright, S. Snyder, K. Harms, Z. Nault, and P. H. Carns, "Umami: a recipe for generating meaningful metrics through holistic I/O performance analysis," *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, 2017.
- [39] C. Xu, S. Byna, V. Venkatesan, R. Sisneros, O. Kulkarni, M. Chaarawi, and K. Chadalavada, "Lioprof: Exposing lustre file system behavior for I/O middleware," in *2016 Cray User Group Meeting*, 2016.
- [40] N. Dryden, R. Böhringer, T. Ben-Nun, and T. Hoefler, "Clairvoyant prefetching for distributed machine learning I/O," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY,

- USA: Association for Computing Machinery, 2021. [Online]. Available: <https://doi.org/10.1145/3458817.3476181>
- [41] N. Tran and D. Reed, "Automatic arima time series modeling for adaptive I/O prefetching," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 4, pp. 362–377, 2004.
- [42] H. Tang, X. Zou, J. Jenkins, D. A. Boyuka, S. Ranshous, D. Kimpe, S. Klasky, and N. F. Samatova, "Improving read performance with online access pattern analysis and prefetching," in *Euro-Par 2014 Parallel Processing*, F. Silva, I. Dutra, and V. Santos Costa, Eds. Cham: Springer International Publishing, 2014, pp. 246–257.
- [43] J. Bez, F. Boito, R. Nou, A. Miranda, T. Cortes, and P. O. Navaux, "Detecting I/O access patterns of HPC workloads at runtime," in *SBAC-PAD 2019 - International Symposium on Computer Architecture and High Performance Computing*, Campo Grande, Brazil, Oct. 2019.
- [44] Y. He, D. Dai, and F. S. Bao, "Modeling HPC storage performance using long short-term memory networks," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2019, pp. 1107–1114.
- [45] D. Li, Y. Wang, B. Xu, W. Li, W. Li, L. Yu, and Q. Yang, "PIPULS: Predicting I/O patterns using lstm in storage systems," in *2019 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS)*, 2019, pp. 14–21.
- [46] R. McKenna, S. Herbein, A. Moody, T. Gamblin, and M. Taufer, "Machine learning predictions of runtime and IO traffic on high-end clusters," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016, pp. 255–258.
- [47] S. Kim, A. Sim, K. Wu, S. Byna, and Y. Son, "Design and implementation of I/O performance prediction scheme on HPC systems through large-scale log analysis," *Journal of Big Data*, vol. 10, no. 1, p. 65, May 2023.
- [48] T. Wang, S. Snyder, G. Lockwood, P. Carns, N. Wright, and S. Byna, "IOMiner: Large-scale analytics framework for gaining knowledge from I/O logs," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 466–476.
- [49] Y. Lu, Y. Chen, R. Latham, and Y. Zhuang, "Revealing applications' access pattern in collective I/O for cache management," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 181–190. [Online]. Available: <https://doi.org/10.1145/2597652.2597686>
- [50] X. Zhang, K. Davis, and S. Jiang, "IOrchestrator: Improving the performance of multi-node I/O systems via inter-server coordination," in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. New Orleans, LA, USA: IEEE, Nov. 2010, p. 1–11. [Online]. Available: <http://ieeexplore.ieee.org/document/5644884/>
- [51] J. Liu, Y. Chen, and Y. Zhuang, "Hierarchical I/O scheduling for collective I/O," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013, pp. 211–218.
- [52] R. Ge, X. Feng, and X.-H. Sun, "Sera-IO: Integrating energy consciousness into parallel I/O middleware," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, 2012, pp. 204–211.
- [53] H. Song, Y. Yin, Y. Chen, and X.-H. Sun, "A cost-intelligent application-specific data layout scheme for parallel file systems," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 37–48. [Online]. Available: <https://doi.org/10.1145/1996130.1996138>
- [54] H. Ather, J. L. Bez, B. Norris, and S. Byna, "Illuminating the I/O optimization path of scientific applications," in *High Performance Computing*, A. Bhatele, J. Hammond, M. Baboulin, and C. Kruse, Eds. Cham: Springer Nature Switzerland, 2023, pp. 22–41.
- [55] M. Casas, R. M. Badia, and J. Labarta, "Automatic phase detection and structure extraction of MPI applications," *The International Journal of High Performance Computing Applications*, vol. 24, no. 3, pp. 335–360, 2010.
- [56] Z. Qiao, Q. Liu, N. Podhorszki, S. Klasky, and J. Chen, "Taming I/O variation on qos-less HPC storage: What can applications do?" in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–13.
- [57] A. Graps, "An introduction to wavelets," *IEEE Computational Science and Engineering*, vol. 2, no. 2, pp. 50–61, 1995.
- [58] A. Tarraf, A. Bandet, F. Boito, G. Pallez, and F. Wolf, "Capturing periodic I/O using frequency techniques [data set]," Zenodo, Feb. 2024. [Online]. Available: <https://doi.org/10.5281/zenodo.10670270>