I/O patterns modeling of HPC applications with call stacks for predictive prefetch

Louis-Marie Nicolas, Salim Mimouni, Philippe Couvée, Jalil Boukhobza

▶ **To cite this version:**

Louis-Marie Nicolas, Salim Mimouni, Philippe Couvée, Jalil Boukhobza. I/O patterns modeling of HPC applications with call stacks for predictive prefetch. Future Generation Computer Systems, 2026, 175, pp.108034. ⟨10.1016/j.future.2025.108034⟩. ⟨hal-05271582⟩

HAL Id: hal-05271582

https://hal.science/hal-05271582v1

Submitted on 22 Sep 2025

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 CC BY 4.0 - Attribution - International License

Future Generation Computer Systems 175 (2026) 108034

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

I/O patterns modeling of HPC applications with call stacks for predictive prefetch

Louis-Marie Nicolas [a,b], Salim Mimouni [b], Philippe Couvée [b], Jalil Boukhobza [a,c,*]

[a] Lab-STICC, CNRS UMR 6285, ENSTA, Institut Polytechnique de Paris, Brest, France
[b] Atos BDS R&D Data Management, Grenoble, France
[c] Institute of Information Science, Academia Sinica, Taipei, Taiwan

ARTICLE INFO

Dataset link: https://github.com/ShinySilver/Graph-Based-IO-Modeling-using-Call-Stacks

Keywords:
Predictive prefetch
I/O prediction
I/O modeling
I/O instrumentation
I/O call stacks
High performance computing
Storage systems

ABSTRACT

Modern high-performance computing (HPC) storage systems use heterogeneous storage technologies organized in tiers to find a compromise between capacity, performance, and cost. In these systems, prefetching is a common technique used to move the right data at the right moment from a slow to a fast tier to improve overall performance while using the costly high-performance tier only when needed. Effective prefetching requires precise knowledge of the application I/O patterns. This knowledge can be extracted through the source code, I/O tracing tools or I/O functions call stacks. State-of-the-art solutions based on the latter approach mainly focus on applications with regular I/O profiles to avoid scalability issues due to the grammar-based techniques used. In this paper, we present an approach based on I/O call stacks that models POSIX and STDIO I/O patterns for both regular and irregular applications, thanks to the use of directed graphs. We present different models usable for prefetching. Our models were used to predict the next I/O call stack on five real HPC applications with a prediction accuracy of up to 98%. Compared to the state-of-the-art Omnisc'IO, they incurred up to 120x lower model overhead (334 ns vs. 45 μs on LAMMPS) and had a model size 10x to 15x smaller (463 B vs. 7 kB on LQCD).

1. Introduction

Multiple storage technologies are used in HPC systems. SSD and NVM are the fastest, but also the most expensive ones [1]. They are usually integrated as burst buffers or caches at the top of the storage hierarchy. HDDs are usually used as the main capacity storage medium. There is more than one order of magnitude difference in latency between the main memory of the compute nodes and the fast storage technology used (fast SSDs and NVM), and another order of magnitude between fast and capacity storage (HDD) [2].

To reduce this performance gap between storage and main memory, multiple techniques were proposed in the literature, amongst which can be found prefetching. Prefetching consists in moving files that are likely to be used in the near future from slower to faster storage devices or a compute node's main memory early enough in order to accelerate future I/Os on these files. For example, such prefetchers are implemented in the Linux kernel [3,4].

One prerequisite for an efficient prefetching technique is knowledge about the I/O patterns of the application. This is especially important because HPC I/O patterns are not just reading at initialization, and writing results and checkpoints [5]. Instead, reading and writing can be done thorough the whole duration of HPC applications [5], which makes the ability to efficiently and dynamically model I/O patterns necessary. The state-of-the-art work related to the extraction of I/O knowledge and prefetching techniques can be classified into 3 categories: (1) Studies of the first category obtain knowledge related to I/O patterns from the source code when available [6,7]. In this **white box** approach, prefetching primitives can be inserted into the source code to load data that are about to be accessed at the right time. Although this approach guarantees that the right data are loaded on time, it makes the assumption that the source code is available and can be modified, which is rarely the case. (2) The second category of solutions considers applications as **black boxes** and relies solely on I/Os issued to make predictions. These solutions may deploy (sometimes costly) tracing tools [8–10] and use probabilistic models [11,12] or sometimes pattern matching techniques [13–16] to model and predict I/O behavior. Although these solutions do not require any prior knowledge about the application, they need to deploy some tracing techniques, and their predictions are not always accurate. (3) Studies from the last

* Corresponding author.
E-mail addresses: louis-marie.nicolas@ensta.fr (L.-M. Nicolas), salim.mimouni@eviden.com (S. Mimouni), philippe.couvee@eviden.com (P. Couvée), jalil.boukhobza@ensta.fr (J. Boukhobza).

https://doi.org/10.1016/j.future.2025.108034
Received 7 March 2025; Received in revised form 15 June 2025; Accepted 23 July 2025
Available online 2 August 2025
0167-739X/© 2025 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

category try to find a middle ground between the two first. They use a **gray box** approach to extract knowledge of applications from their intercepted call stacks to model I/O patterns. Several studies [17–19] compress this I/O call stack using the Sequitur [20] algorithm to form a grammar model used for predictions. We consider that application source code is often unavailable, which is why we do not focus on the white-box approach. Moreover, traditional black-box approaches either use traces that are expensive to produce and store [8], or use pattern matching whose prediction cost scales with the number of patterns [13], limiting the number of patterns that can be learned at the same time. Furthermore, state-of-the-art studies [17,18] have shown that call stack information could be used to predict I/Os, without the need to store traces through the use of a grammar model with only a few limitations. For these reasons, in this paper we focus on the use of a gray-box approach.

One of the main issues with state-of-the-art gray box approaches is that the compression algorithm encodes the I/O call stack sequence in a lossless way. This makes it possible for the whole I/O call stack sequence to be reproduced from the grammar model. Although, as justified in Omnisc'IO [17,18], this is relevant when modeling applications with regular, deterministic I/Os, this method shows its limits on applications with irregular I/Os. This is because the grammar would grow rapidly and the search for the right I/O to prefetch would be less accurate and would last much longer. We observed that in some real HPC applications, state-of-the-art predictors could take more than 10 ms per I/O prediction for a poor accuracy (around 32% in some cases).

In this paper, we aim at creating a fast, low-overhead application I/O prediction model with the ability to forecast, with good accuracy, the I/Os for both regular and irregular applications. For this sake, we present *GrIOt*, a **Gr**aph-based Modeling of **I/O** call s**t**acks for Predictive Prefetch. We believe that being able to reproduce losslessly the I/O call stack sequence is not necessary to make accurate predictions. As a consequence, GrIOt relies on building directed graphs from I/O call stacks. In such a graph, each node consists of one (or a sequence of) I/O call stack(s), and each edge is a transition from one (or a sequence of) call stack(s) to another. The size of such a model scales primarily with the number of unique call stacks and subsets of call stacks but is not directly affected by the number of I/Os or by the regular nature of the application's I/Os. While state-of-the-art models compress the entire call stack sequence into their models, our graph model only saves transitions, which makes it much more scalable.

Predicting the next I/O in the created graph consists in choosing amongst the outgoing edges of a node the one that is the most likely to be used by the application. To make this choice, we propose two simple heuristics: Most Recently Used Edge (MRU), and Most Frequently Used Edge (MFU).

In our previous work [21], GrIOt created one graph for each application process. This design is limited to the prediction of the next I/Os of each application process, with no way to selectively predict the next I/Os of a specific file rather than making prediction for a whole process, which in case of interleaved I/Os can make predictions harder to use for prefetching.

In this paper, we show that instead of creating one model per application process, it is possible to create one model per opened file and to make prediction per file, which is more suitable as common prefetching algorithms run per file [11]. Moreover, this alternative granularity emphasizes file temporal and spatial locality, being able to predict the most recently or frequently used edge of a specific file rather than for the whole application process. Finally, in this paper, we characterize the call stack instrumentation methods and parameters in more depth than the existing literature.

Our contribution was evaluated on a real cluster with five state-of-the-art HPC applications.

Depending on the heuristic or the modeling granularity selected, our models can offer a drastically reduced overhead up to 120x lower compared to state-of-the-art OmniscIO [17,18] for a single I/O prediction,

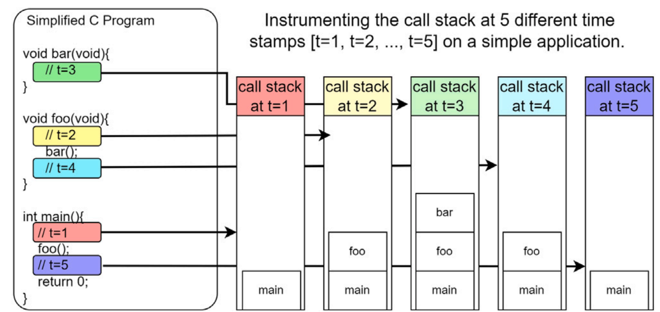

Fig. 1. Representing the call stacks at different times on a simple application.

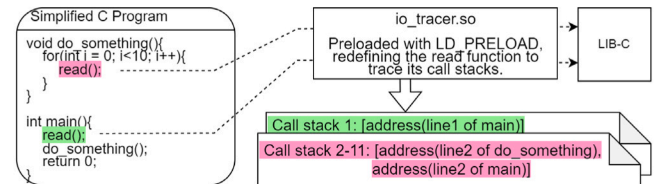

Fig. 2. Intercepting the call stacks of an I/O function.

while providing a similar or better I/O prediction accuracy of up to 98% and a lower model size up to 100x smaller.

In the 2nd Section we introduce some basic concepts about call stacks and their instrumentation, then our contribution is described in the 3rd to 6th Sections. The 3rd Section is an overview of GrIOt. The 4th Section presents how we trace I/Os and preprocess I/O call stacks, the 5th presents the GrIOt modeling algorithm, and the 6th presents the GrIOt prediction algorithm using heuristics. In the 7th Section, we evaluate the proposed strategy. At last, we present in the 8th Section some related work and then conclude.

2. Background on call stacks

The call stack is a per process memory buffer in which the system stores the return address each time a function is called on a Last In First Out basis. When the execution of the function ends, the return address at the top of the call stack is used (and then removed) to execute the next instructions.

This is illustrated in Fig. 1, in which the call stack we would obtain by calling an instrumentation function is represented over the duration of a simple application. If we focus on the call to the function *bar*, we can see that between$t = 2$ and t=3, during the function call, a return address to *bar* was added to the call stack. This return address was then removed (and used) between$t = 3$ and$t = 4$ at the function *bar* return time, enabling returning to the previous function.

A snapshot of the call stack can be obtained using the $backtrace(void * buffer[.size], int size)$ function from the GNU C extension, or through the use of the *libunwind* library. In both cases, the call stack instrumentation function takes as a parameter the maximum number of return addresses to collect from the top of the stack.

It can be noted that because the instrumentation of the call stack is done through a function call, the address of the current function is added for a short time to the top of the call stack. This is why the call stack that was instrumented at $t = 1$ in Fig. 1 contains a return address to *main*.

LD_PRELOAD is an environment variable available on Linux and other systems that can be set to load a specified shared library before any other, including the C standard library. This makes it possible to intercept specific function calls, as well as to execute specific actions when those calls are being intercepted.

```c
#include <stdio.h>
#include <dlfcn.h>

// 1: Obtaining a pointer to the original read function
ssize_t (*iotracer_safe_read)(int fd, void *buf, size_t count)
    = dlsym(RTLD_NEXT, "read");

// 2: Intercepting calls to the read function with all parameters
ssize_t read(int fd, void *buf, size_t count){
    ssize_t return_value = iotracer_safe_read(fd, buf, count);
    // 3: You can call backtrace, update a model, or make a prediction here.
    fprintf(iotracer_trace_file, "Write something, do something.");
    return return_value;
}
```

Fig. 3. Intercepting the read function with LD_PRELOAD and *dlsym*.

For example, creating a function called *read* in a shared library and loading this dynamic library through LD_PRELOAD will make all the calls to the original Lib-C POSIX *read* function be replaced by our own shared *read* function. If, at a given point, we need our own version of the *read* function to call the standard C *read* function, we can rely on the POSIX *dlsym* call. We show in Fig. 2 an example with a tracer that uses both the LD_PRELOAD and *dlsym* function to override the *read* function call, store the call stack snapshots related to those calls, and then call the original Lib-C POSIX *read* transparently.

Finally, a possible C implementation of such a tracer is proposed in Fig. 3. This very short code can be compiled as a shared library, and will, when loaded through LD_PRELOAD, intercept all dynamic calls to the above-mentioned *read* function, and call the original *read* function obtained through *dlsym*. After the first comment of the code snippet, a pointer to the original *read* POSIX function is obtained. Under the second comment, a function with the same name, return type, and parameters as the original *read* POSIX function is defined. Because it shares the same name, this function will be called instead of the original *read* function if the library is loaded with LD_PRELOAD. Finally, with the third and last comment, at interception time the original *read* function is called in order to execute the I/O of the user code, and we log the function call to a trace file. Because our function shares the same parameters as the original function, all the I/O function parameters are available during interception and can as such be used for tracing and modeling.

3. GrIOt overview

As illustrated in Fig. 4, the GrIOt strategy can be decomposed into three major steps:

- **Intercepting I/Os**: I/O function calls are intercepted per process using a custom user-space I/O tracer (step I).
- **Modeling I/Os**: The I/O function calls and their corresponding pre-processed call stacks are captured in a directed graph (step II). This graph is created and continuously updated during the application execution as long as GrIOt is ran. Metadata, such as the frequency and recency of used edges, are similarly dynamically updated, and saved according to the online prediction heuristic used. In order to be able to predict the next I/O of each application process, we created a graph representation of the call stacks based on three different granularities. The first option is to create one call stack graph per process to get the full per process I/O pattern. The second granularity is to create one graph per opened file (on a per process basis). By doing so, it is possible to predict the next I/Os of a specific file, which is more relevant in a context of prefetching. However, it does not enable model reuse as models are discarded when files are closed. To avoid this issue, rather than creating one model per opened file, with the third granularity we create one model per call stack of a file-opening function, enabling model sharing and reuse between files opened with the same function. That way, I/O patterns do not have to be relearned every time a file is opened. Overall, when files opened with the same function call stack exhibit different I/O behaviors, the per-open modeling granularity is the most appropriate, as it provides isolated models for each file instance. Conversely, the per-open-call-stack modeling granularity is best suited when files opened with the same function call stack exhibit similar I/O behavior. In this case, a single model can be shared across such files, improving efficiency. Moreover, this granularity is particularly advantageous when files are opened and closed for only a few I/Os, as it retains models associated with opening call stacks and enables accurate predictions from the first I/O—unlike the per-open granularity, which starts from an empty model for each new file.
- **Online prediction**: I/O predictions are performed from the built graph following two heuristics (step III): Most Recently Used Edge (MRU) and Most Frequently Used edge (MFU). Some hash table mechanisms are used by GrIOt to accelerate the graph search and the identification of I/O parameters. The choice of the heuristic has side-effects regarding the modeling: Most Frequently Used edge requires adding weights to the edges of the graph, while Most Recently Used edge requires saving the latest edge followed for each node as a metadata. Moreover, the choice of the modeling granularity introduced above has an impact on the metadata locality, and as such on the heuristics. For instance, selecting the per open call stack modeling granularity means MRU will select the most recently used edge relative to the target opened file, rather than the most recently used edge of the whole application.

Algorithm 1 GrIOt modeling and prediction algorithm

1: **function** ON_IO_INTERCEPTED(*io*, *context_size*, *prediction_heuristic*)
2: *#### (I) Preprocessing call stacks*
3: Obtain, preprocess, and hash the I/O call stack
4: *#### (II) Modeling I/Os*
5: Update the new context
6: If it does not exist, create a node or edge in the graph using the new context.
7: Update the graph recency or frequency metadata, depending on *pred_heuristic*
8: Update *I/O hash table* for the new I/O call stack hash.
9: *#### (III) Online Prediction*
10: **if** outgoing edge count of current node == 0 **then**
11: **return** *The call stack of the current node*
12: **else if** outgoing edge count of current node == 1 **then**
13: **return** *The call stack of the sole next possible node*
14: **else return** *Call stack selection using* prediction_heuristic

These steps are performed **online** every time the application performs an I/O function call. Alg. 1 shows the sequencing of the steps of GrIOt, it is referenced all through the explanations.

4. GrIOt step 1: Intercepting I/Os and preprocessing call stacks

The first step of GrIOt, as illustrated in Step I of Fig. 4, is the interception of the dynamically linked POSIX and STDIO I/O functions (such as `read`, `pread`, and `pread64`) and the preprocessing of the call stacks. This interception is done through the use of LD_PRELOAD introduced previously. Our tracer is custom-made as, to the best of our knowledge, no existing tracers were designed to intercept and store I/O call stacks at the time of this publication.

As illustrated in Fig. 5 step (I), our custom tracer starts by intercepting dynamically linked I/Os. When the user application wants to call an I/O function, a similarly named function in our tracer is called instead. In this function, the original I/O function is called as well as the C backtrace function (II) to obtain the call stack. As introduced in the background section, this raw call stack consists of a list of return addresses. This includes the I/O function call that was intercepted, that is "*read*" at line 2 of main in the example code of Fig. 5.

It has to be reminded that the return address for a given function is not the same from one execution to another due to address space layout randomization [22], a well-known technique used to mitigate memory corruption vulnerabilities.

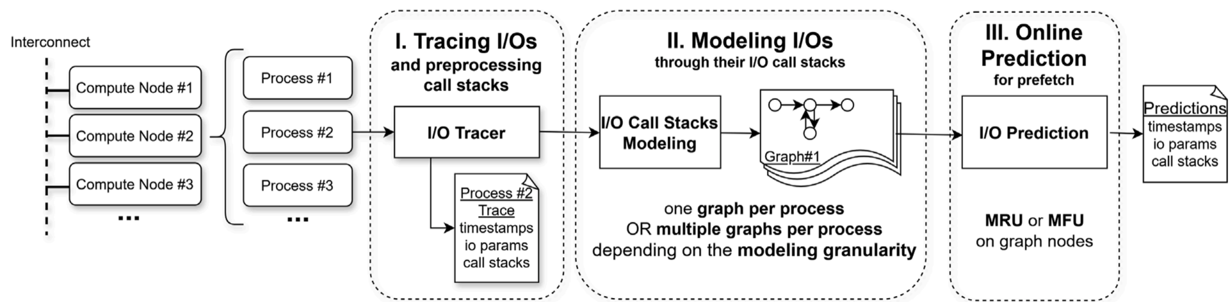

Fig. 4. GrIOt overview.

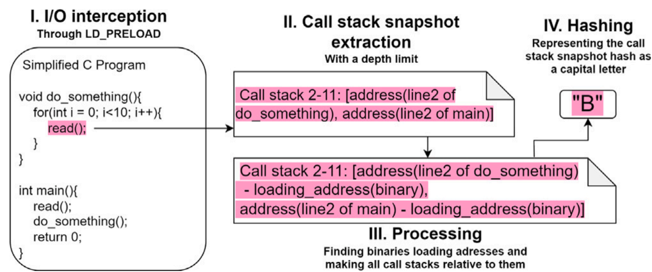

Fig. 5. Call stack preprocessing.

In fact, existing studies using call stacks focus on online prediction and do not consider reusing an application model [17–19], which makes their reuse challenging. We argue that while the exact number of HPC applications may vary considerably depending on the specific field or context, it is common to find a handful of key applications being run on a routine basis, for example in the weather forecasting field where the same forecasting applications are usually run routinely for extended periods of time [23]. That is why we consider that being able to reuse application models could be of interest.

In order to make the traced I/O call stacks comparable from one process to another, we introduced step (III) of Fig. 5. The return addresses that are contained in the call stack snapshots are made re-usable from one run to another. This is achieved by making the return addresses of the call stack snapshots relative to the loading address of the binary that is available in the $/proc/[pid]/maps$ file on Linux. Doing so, a given call is always identified by a constant address.

Finally, in step IV of Fig. 5, GrIOt computes a unique identifier for every unique call stack snapshot through the use of a general-purpose noncryptographic hash function, a 64 bits implementation of Murmur 2 [24]. This makes it easier to check whether two call stack snapshots are equal and later makes searching the graph for a given call stack faster. We refer to these unique call stack identifiers as "call stack hashes".

In the remaining figures as in step IV of Fig. 5, each stack snapshot is addressed by its call stack hash, which is represented by a capital letter for ease of use. Similarly, a sequence of I/O call stack snapshots is represented as a sequence of capital letters.

After going through these four steps, the GrIOt tracer possesses not only the I/Os parameters that are easily obtainable as illustrated in Fig. 3, but also a re-usable I/O call stack hash. This information, as illustrated in Fig. 4, can be exported as a trace file. However, doing so creates an additional overhead on the storage device. Since GrIOt is an online algorithm, this feature is disabled by default, and the I/O information is simply passed to the modeling algorithm without being stored. However, if one needs to capture I/O traces and call stacks for further analysis, GrIOt could be triggered to do so.

We note that while our tracer does not intercept calls to libraries such MPI-IO or HDF5, it will still correctly intercept I/Os when the libraries themselves call POSIX or STDIO functions. It cannot be used yet, however, to instrument memory-mapped files. Moreover, the call stack might not be reliable for programs written in interpreted languages, such as Python.

5. GrIOt step 2: Modeling I/Os

At the end of step I of Fig. 4, a call stack snapshot hash and associated I/O parameters are produced. In step II, that is the I/O modeling phase, these informations are used to create a model made of two parts:

- An I/O call stack directed graph. It is tasked with modeling the I/O call stack sequence for I/O pattern representation.
- An I/O parameter table. It is tasked to associate I/O parameters such as I/O size and offset to each unique I/O call stack, which are required for the model to be used for predicting I/O parameters to use for prefetching.

Together, these two elements can be used to model an I/O sequence. They are detailed in Sections 5.1 and 5.2 below.

This I/O modeling can be done at multiple levels. State-of-the-art work focuses on creating one I/O model per process. However, this modeling granularity only enables the prediction of the next I/Os of the application process. In order to be able to selectively predict the next I/Os of a target file, we propose two alternative finer-grained, per file modeling granularities. The three modeling granularities supported by GrIOt are thus the following:

- One model **per process**. One call stack graph is created per process. The model can predict the next I/Os of the current process.
- One model **per open**. One new model is created for every new file descriptor. Unlike the previous granularity, this modeling granularity enables per file prediction and metadata.
- One model **per open call stack hash** also called **per open-hash sometimes for short**. This modeling granularity similarly to the previous one enables per file prediction and per file metadata. Unlike the previous granularity, the model is saved when the file descriptor is closed, enabling memory of the previously opened files.

The design and implementation of these modeling granularities is detailed in Section 5.3.

5.1. I/O call stack directed graph

The main element of the GrIOt model is a directed graph that models the sequence of I/O call stacks. We propose two different graph modeling solutions:

- Naive modeling solution: one node = one call stack snapshot hash.
- Advanced solution: one node = one call stack snapshot hash and a history of the last few previous call stack snapshots.

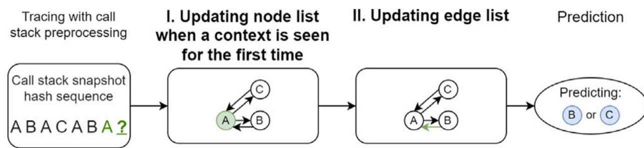

Fig. 6. Online update of a graph with context size=1.

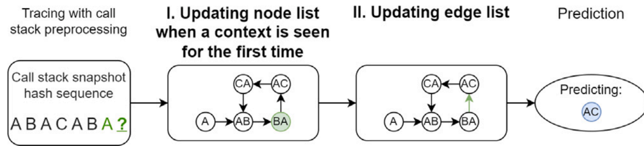

Fig. 7. Online update of a graph with context size=2.

The naive approach to modeling I/O call stack snapshot hashes would be to create a directed graph, in which a node would be a unique call stack snapshot hash (that is a unique I/O function call). An edge is created between two nodes in run-time to indicate that the two subsequent I/O functions call stack were called in order. This is the first modeling solution we consider.

This model, while straightforward to build, has some real issues. For example, if a given call stack hash is followed by two or more different call stack hashes in the graph (several outgoing edges), the prediction is more difficult to infer.

Such a case is illustrated in Fig. 6. In this example, a sequence of call stack snapshot hashes is created: "ABACAB". Then, when the last call stack hash is "A" (green in the Figure), the predictor has no clue of knowing which one of "B" or "C" node to follow as there are two outgoing edges. Moreover, this model does not consider the fact that the next possible I/O call stack hashes might not necessarily be solely determined from the last call stack hash.

Previous issue can be solved by creating graph nodes that contain more information about the history of an I/O in order to more easily operate a choice when several outgoing edges exist. This is the second modeling solution we consider. To do so, the model may store in the graph node not only the call stack snapshot hash of the captured I/O, but also the last few ones preceding it. The node's call stack hash plus the stored history is called the **context** and the number of call stack hashes per node the **context size**.

For example, in Fig. 7, a context size of two is chosen, which means that a node contains both the current I/O function call stack snapshot hash plus the preceding I/O's. Due to this change, the node that was called "A" in Fig. 6 is split into two nodes "BA" and "CA", that is, the node containing the I/O related to call stack "A" that was preceded by "C" and the one that was preceded by "B", respectively (the last letter represents the current hash and the ones preceding it its history). Due to this, an edge between "BA" and "AC" means the I/O call stack sequence "BAC" was observed, which also means that after an "A" that was preceded by a "B", "C" is a possible prediction. Using this graph, we observe that it is now possible to predict the next call stack without ambiguity.

The larger the context size, the more precise the model is. However, adding history to each node has two impacts: (1) the node size grows as it should store more call stack snapshots, and (2) the graph size grows as it may contain more nodes. As a consequence, there is a trade-off to strike to find the best context size for a given workload. This is evaluated in the experimental part.

5.2. I/O parameters table

While the I/O call stack graph models the I/O call stack sequence, the I/O parameters table makes it possible to associate I/O parameters to the I/O call stacks. Together, they can model both the I/O sequence and parameters.

Each call stack snapshot is captured along with the I/O parameters of the related I/O function. These are stored in a table (one line per unique snapshot) and contain all the metadata that can be extracted from the intercepted function call, as illustrated in Fig. 3, that is: the file identifier, the offset, the I/O size and the inter arrival time (with regard to previous I/O function). To associate one line of parameters to a graph node, we created an **I/O hash table** whose keys are call stack hashes and whose values are the I/O parameters last seen for these call stack hashes.

This I/O parameter table is based on the assumption that I/Os with the same call stack will have similar I/O parameters. This assumption was first made in Omnisc'IO [17,18].

5.3. Modeling granularity and algorithm

In state-of-the-art studies, regardless of the nature of the model, a single model is created per process. The rationale behind this option is that the I/O patterns are issued on a process basis. In this paper, we introduced two additional modeling granularities. The algorithm and data structure presented before in this section are applicable to all three granularity levels, with the following small differences in the number of graphs, parameter tables and associated data structures:

- one model **per process**: there is one single graph and parameter table per process. No additional data structure is needed.
- one model **per open**: one model is created per file descriptor. There is a hash table associating one graph and one parameter table to each unique file descriptor in a near-constant time. Two different file descriptors cannot share a graph or parameter table. As such, this modeling granularity enables per file prediction. As a side effect, prediction metadata such as recency and frequency are local to each opened file. One new model is created every time an "open-like" function is called. Models are discarded when file descriptors are closed.
- one model **per open call stack**: there is a hash table associating one graph and one parameter table to each file descriptor in a near constant time. Multiple file descriptors can share the same graph. Moreover, there is another hash table similarly associating one graph and one parameter table to each open call stack in a near constant time. Two different open call stacks cannot share graph nor a parameter table. The main difference with the previous modeling granularity is that the model is shared between file descriptors that were opened with similar open call stack hash, and the model is not discarded until the end of the process. So there is a memory effect on previously opened files.

Aside from these granularity-specific implementation details, the GrIOt algorithm is described in Alg. 1. We detail here the modeling steps of this algorithm, namely the update of the context stack, graph and table :

Updating the context stack. When an I/O is intercepted and its call stack processed, one prerequisite to the creation of the node in the graph is the context update, see line 5 of Alg. 1. The context is kept track of through a stack whose size is equal to the context size.

Updating the graph. After updating the context stack, a graph node identified by this context is created in case it does not already exist, see line 6 of Alg. 1. Then, if an incoming edge does not exist between this node and the previous node, it is added accordingly. In order to check if a node already exists in the graph in nearly constant time, a hash map is used to store all the nodes in the graph. In order to check if a node has an edge to another node, every node contains its outgoing edges in an array. While this operation requires iterating over the array, an array is more memory efficient and just as fast as a hash map because most nodes have less than a dozen outgoing edges.

Updating the graph metadata. After updating the graph, recency and frequency data are updated. Their addition is not trivial. A naive implementation would add these values to the graph edges. However, when using the *per open call stack* model granularity, we identify an edge case: multiple file descriptors can be sharing a graph at the same time. For the recency and frequency to be used effectively, locality is necessary. Having multiple file sessions updating the same metadata in the same graph would break this locality. As such, the recency and frequency of another file sharing the same graph is not pertinent. That is why the graph metadata should be a separate data structure from the graph. That way, when using the *per open call stack* model granularity, the metadata granularity used for prediction is *per file* rather than *per open call stack*, thus respecting recency and frequency locality.

Updating the table. Finally, the I/O hash table is also updated with the I/O parameters relative to the related intercepted I/O function. Another costly operation is the search for the I/O parameters in the table once a node has been identified for prefetching. This was also addressed thanks to a hash map.

6. GrIOt online prediction

The last step of GrIOt as presented in the step III of Fig. 4 is the online prediction of I/Os. It can be decomposed into two substeps: predicting the next I/O call stack, and using this I/O call stack to predict the parameters of the next I/O. These two substeps are detailed below.

6.1. I/O call stack prediction

I/O prediction is the last step of GrIOt, after the I/O interception and I/O modeling steps. At this step, our prediction algorithm is using the model, a context value that can be associated with the current node in the graph.

As illustrated in line 10 of Alg. 1, GrIOt prediction algorithm first checks if there are any outgoing edges from the node corresponding to the current context. If none is found, which happens when a call stack is seen for the first time, we make the assumption that the call stack will repeat itself (same function call) but with an update on the parameters corresponding to a sequential access with the same I/O size. This is an extreme case, as it will only happen the first time a context value is seen.

If there is a single outgoing edge, GrIOt predicts in line 13 of Alg. 1 that the call stack corresponding to this edge will be the next node in the graph.

Finally, if there is more than a single outgoing edge, we in see line 14 of Alg. 1 that GrIOt uses a heuristic to choose the next node for prediction purposes. In order to leverage I/O locality [12], we propose two simple strategies, Most Frequently Used (MFU), and Most Recently Used (MRU) edge:

- *Most Frequently Used Edges (MFU)*. To use this heuristic, GrIOt needs to save information related to the frequency of each outgoing edge of each node. To do so, we added frequency weights to the edges. All edges have a default weight of 1. GrIOt increments a weight when it is traversed, while the weights on the other edges of the node are decremented. We used this to approximate frequency.
- *Most Recently Used Edges (MRU)* MRU is used as an alternative to MFU because it can leverage temporal locality. The edge that was last traversed is saved for each node.

GrIOt can predict several I/O call stacks by traversing the graph using the above algorithm, using its own prediction to update the context stack.

6.2. I/O parameters prediction

Once the next graph nodes are predicted, they can be translated into an I/O request through the I/O parameters table stored for each call stack snapshot as introduced in Section 5.2. We used an I/O table similar to the one used in Omnisc'IO [17,18].

7. Experimental evaluation

7.1. Methodology

GrIOt was implemented in C. CMake and compilation-time macros were used to create one GrIOt binary per modeling granularity. These binaries can perform all three steps of GrIOt by themselves: I/O interception, online I/O modeling, online I/O prediction. In addition to GrIOt three steps, we add an exporting step at the end of the execution of a traced process in which the GrIOt graph and evaluation metrics are written as csv.

Real HPC applications were run with GrIOt. They are introduced in more details in Section 7.2.

Three sets of experiments were performed. The first set compares GrIOt with state-of-the-art studies. The second set compares different configurations of GrIOt. The third set studies in more depth the overhead of call stack instrumentation.

GrIOt was evaluated against the Sequitur-based Omnisc'IO [17,18, 25]. We have chosen Omnisc'IO for this evaluation because it is the latest and most efficient work we found using a model based on call stacks for I/O prediction and with a code that is publicly available.

The cluster we used in this evaluation is composed of 32X440-A5 BullSequana nodes, with two AMD EPYC Rome 7282 CPU (16 Cores, 2.8GHz-120 W) and 128 GB of DDR4-3200 RAM per node.

7.2. Traced applications

As introduced in Table 1, five real HPC applications were traced.

LAMMPS [26] is the name of a molecular dynamics tool. It stands for *Large-scale Atomic/Molecular Massively Parallel Simulator*. We use a publicly available input file[1] to simulate a *1WDN Glutamine-Binding Protein* with 1024 processes on 14 compute nodes.

NAMD [27] is similarly a molecular dynamics tool. We use another publicly available input file to run NAMD on a STMV virus.[2]

Incompact3D [28] is a Navier–Stokes solver. We used a publicly available input file, the X3D benchmark input file.[3]

Nemo is a framework for ocean simulation and climate studies. We used a privately available input file.

LQCD is an HPC application for the simulation of Lattice Chromodynamics. We used a privately available input file.

Aside from a short description of the applications, Table 1 also contains metrics related to the applications' I/O call stacks. They can be used to get a measure of the I/O call stack patterns complexity. Most notably, a low complexity of the I/O call stack patterns can be translated into (1) a higher percentage of repeating call stacks (up to 97.91% on LQCD) or (2) a low number of unique call stack transitions compared to the total number of unique call stacks (only 110 unique transitions for a total of 85 unique call stack on Incompact3d).

1 https://www.hecbiosim.ac.uk/access-hpc/benchmarks

2 https://www.ks.uiuc.edu/Research/namd/utilities/stmv/

3 https://github.com/xcompact3d/X3D-benchmarking

Table 1
List of the applications used in the evaluation.

Application	Description	Nodes	Processes	I/O volume and count	# unique call stacks	# unique call stacks transitions	% of repeating call stacks
NAMD	Molecular Dynamics, 1.1M Atoms: STMV 210M	12	12	81.5GB 93.3e3 I/Os	371	718	9.32%
LAMMPS	Molecular Dynamics, 10k Atoms: 3NIR Crambin	14	896	13.1GB 18.5e3 I/Os	39	52	80.18%
Incompact3d	Navier–Stokes solver	10	640	13.8GB 911e3 I/Os	85	110	0.28%
LQCD	Quantic chromodynamics	16	3072	73.0GB 5.11e6 I/Os	319	643	97.91%
NEMO	Ocean simulation	8	256	22.8GB 143e3 I/Os	229	312	32.52%

7.3. Evaluation metrics

We identified five evaluation metrics. The performance of the model was evaluated using two metrics. The first is the accuracy of the stack prediction for the next call. Since we focus, in this work, on evaluating the prediction for a prefetch purpose, we also evaluated the percentage of I/O volume whose I/O call stack was correctly predicted. While the first metric gives an idea about the number of I/Os correctly predicted, the second one gives the volume of I/Os correctly predicted.

The cost of our model is evaluated through three metrics: the instrumentation overhead, model overhead and model size. Model overhead and model sizes are discussed in experiments 1 and 2, while instrumentation overhead is exclusively discussed in the third experiment, as the state of the art is using a technique similar to ours for instrumentation.

7.3.1. Prediction accuracy

A call stack prediction is considered correct if it matches exactly the next real I/O. The prediction accuracy is the percentage of correct predictions. We also used the weighted prediction accuracy, that is, the percentage of the total I/O volume whose I/O call stack was predicted correctly.

7.3.2. Overhead

We measured the duration of the applications cited in Section 7.2 with and without GrIOt. The delta between these two durations is what we call the **total overhead**.

Then, we measured the duration of these applications with a build of GrIOt that only does the interception step. The delta between this duration and the duration without interception is called the **instrumentation overhead**.

The **modeling and prediction overhead** can be obtained by subtracting the instrumentation overhead from the total overhead.

In order to produce these detailed overhead metrics for Omnisc'IO, we ran Omnisc'IO's Omnilizer binary [25] on a trace of each application. By doing so we obtained its modeling and prediction overhead. However, we were not able to use Omnisc'IO tracer module to measure its instrumentation overhead. As Omnisc'IO is using LD_PRELOAD, similarly to GrIOt, we considered that it should have the exact same instrumentation overhead.

Then, we ran GrIOt a third time, this time enabling model and statistics dumping through the environment variables detailed in Section 7.1. We obtained, amongst other metrics, the I/O count. The **Overhead per I/O** can be obtained by dividing the overheads previously measured on each application by the I/O count. This metric is critical to the future integration of GrIOt with a prefetcher, as the cost per I/O introduced by GrIOt must not exceed the performance gains provided by the prefetcher; otherwise, overall application performance would be degraded rather than improved.

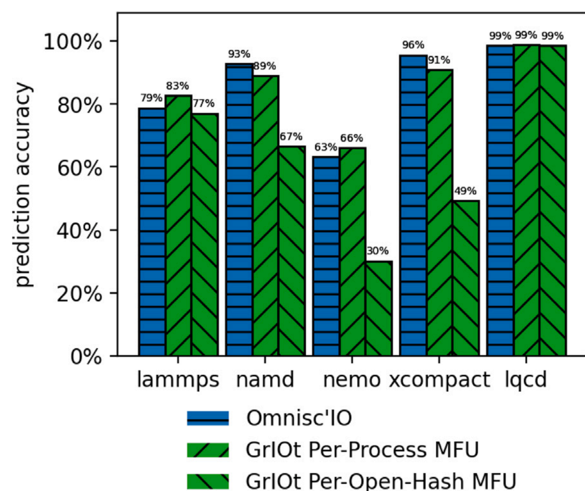

Fig. 8. Comparison of the prediction accuracy of GrIOt and Omnisc'IO.

7.3.3. Model size

We extracted the size of the model from the GrIOt dump. It is not the average size of all the models in a process, but rather the memory footprint of GrIOt per process, that is the sum of the sizes of all the call stack graphs, graph metadata and context stacks of a process. For Omnisc'IO, Omnilizer binary outputs the total number of lines in its grammar model. We modified it to output a size, so we have a comparable metric.

7.4. Results discussion

Three sets of experiments were performed. The first set compares GrIOt with existing studies. The second set tests different configurations of GrIOt. The third one evaluates the instrumentation overhead.

7.4.1. Comparing GrIOt with prior work

We performed a first set of experiments to compare our model with Omnisc'IO.

Prediction accuracy: In Fig. 8, the prediction accuracy of GrIOt and Omnisc'IO is compared. Two GrIOt configurations were used, and were the same for all the following 4 paragraphs. The first one uses the per process modeling granularity, while the second one uses the per open call stack hash modeling granularity that was newly introduced in this paper. Both use MFU as the prediction heuristic, with a context size of 2. It can be observed that GrIOt with this baseline configuration has a prediction accuracy comparable to that of the state-of-the-art study. GrIOt with per process granularity has very similar performance to Omnisc'IO: it is doing slightly better on 3 workloads and slightly lags behind for the three others. This validates our approach: GrIOt lossy I/O

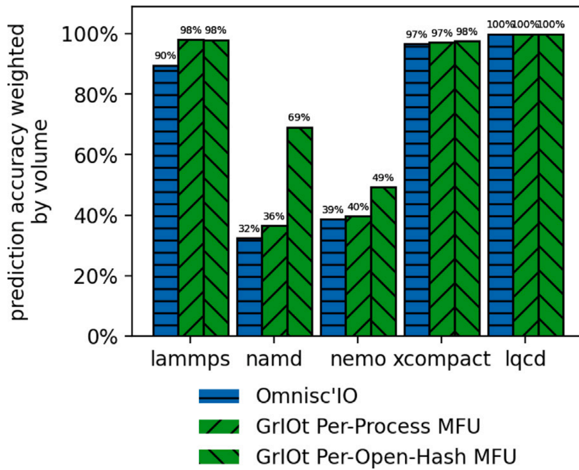

Fig. 9. Comparison of the weighted prediction accuracy of GrIOt and Omnisc'IO.

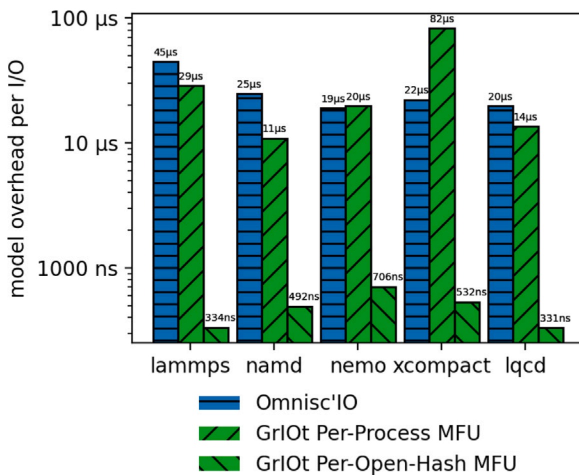

Fig. 10. Comparison of the average modeling and prediction overhead per I/O of GrIOt and Omnisc'IO.

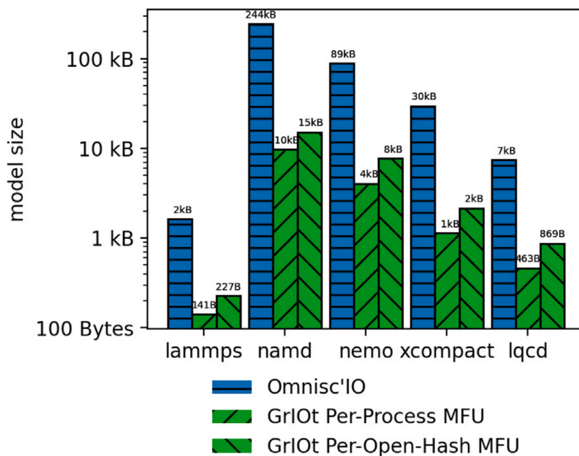

Fig. 11. Comparison of the model size of GrIOt and Omnisc'IO.

call stack sequence modeling can offer similar accuracy to the lossless encoding of the I/O call stack performed by Omnisc'IO. However, GrIOt with per open call stack hash granularity has lackluster accuracy. This can be explained by the fact that having information regarding all the files at the same time (per process option can be useful for prediction,

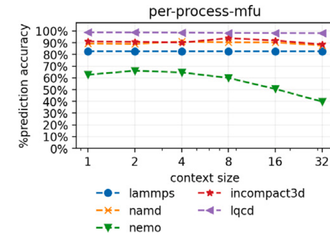

Fig. 12. Effect of context size on accuracy on GrIOt.

especially for predicting log files, which makes for a lot of small I/Os. This useful information is used with GrIOt per process, while it is not used for GrIOt per open call stack hash since the model is per file.

Weighted prediction accuracy: In Fig. 9, the weighted prediction accuracy is used as an alternative accuracy metric to compare GrIOt and OmniscIO. Rather than evaluating the number of I/Os, here we evaluate the correctly predicted volume of I/Os. We can see a large change in the accuracy values when it is weighted with the I/O volume. This is because while Omnisc'IO and GrIOt are predicting correctly a number of I/O call stacks, the volume of these I/O is not necessarily high. Most notably, writing log files and reading parameters is often done through many very small I/Os. Because of their high number, these I/Os can have a significant impact on the accuracy metric despite their relatively low I/O volume. We can see that GrIOt with our newly introduced per open call stack hash modeling granularity is doing better than both Omnisc'IO and GrIOt with the per process modeling granularity. That is because by creating file-level model with the per open call stack hash granularity, multiple simple, easy-to-predict models are created rather than a single complex monolithic per process model with more complex predictions.

Model overhead: In Fig. 10, the prediction and modeling overhead per I/O of our model and Omnisc'IO are compared. The model overhead is the overhead of updating the model after an I/O and predicting the next one. This metric does not comprise the interception cost, which is the same for both GrIOt and Omnisc'IO. It can be observed that the per open call stack hash model has a much lower overhead per I/O than Omnisc'IO, with a reduction of up to 120x on LAMMPS. Unlike a grammar model, which requires a complex algorithm for prediction, making prediction in a graph given a node and a heuristic is straightforward and has a low complexity. However, GrIOt-per process suffers slightly worse performance. This can be explained by the single large graph that needs to be traversed (rather than multiple smaller graphs for the per open call stack hash option), and a lackluster hash map implementation. It still has a much smaller overhead than Omnisc'IO in most cases, the only exception being Xcompact3d. We think that this higher overhead is due to hash collisions in GrIOt hash maps reducing the performance. As such, better results could be obtained by expanding the size of the hashmap, or trying an alternative hash function.

Model size: In Fig. 11, the model size of both our model and Omnisc'IO are compared. **The size of our graph-based model is consistently lower than the size of the grammar-based model of Omnisc'IO,** up to 100x lower on LQCD. This can be explained by the fact that unlike Omnisc'IO, GrIOt does not try to encode the I/O call stack sequence losslessly. We note that the per open call stack hash granularity has slightly higher model size than the per process granularity: this can be explained as it maintains multiple smaller graphs rather than a single large graph, which is more costly due to the need for some additional data structures.

7.4.2. *Exploring GrIOt parameters*

In the previous experiment, we performed a comparison of GrIOt and state-of-the-art work. In this second set of experiments, we evaluated the impact of context size, model granularity, and heuristic choice on accuracy, model overhead, and model size (see Table 2).

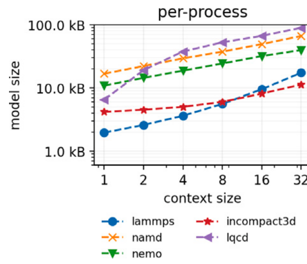

Fig. 13. Effect of context size on model size.

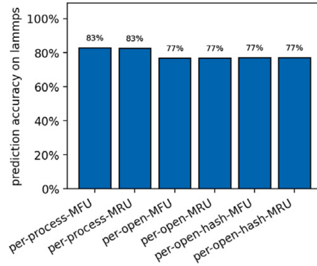

Fig. 14. Effect of model granularity and prediction heuristics on prediction accuracy for LAMMPS.

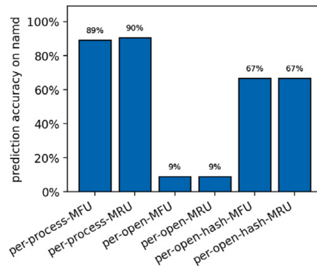

Fig. 15. Effect of model granularity and prediction heuristics on prediction accuracy for NAMD.

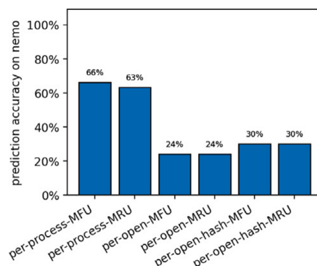

Fig. 16. Effect of model granularity and prediction heuristics on prediction accuracy for NEMO.

Table 2
Effect of model granularity on GrIOt model size.

	LAMMPS	NAMD	NEMO	Incompact3d	LQCD
Per-Process	141 bytes	9.8 kB	4.1 kB	1.1 kB	463 bytes
Per-Open-Hash	227 bytes	15.2 kB	7.8 kB	2.2 kB	870 bytes
Per-Open	281 bytes	2.4 kB	11.9 kB	383 bytes	283 bytes

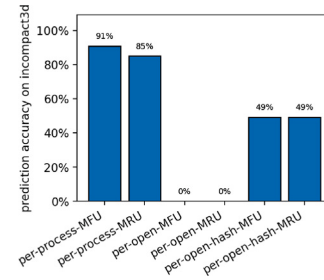

Fig. 17. Effect of model granularity and prediction heuristics on prediction accuracy for Incompact3d.

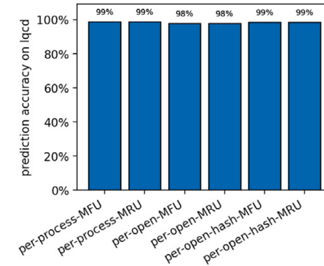

Fig. 18. Effect of model granularity and prediction heuristics on prediction accuracy for LQCD.

Effect of context size : In Fig. 12, the prediction accuracy of GrIOt is shown as a function of context size. The model granularity and prediction heuristic is fixed to our baseline, per process with MFU. We can see that for some applications, especially NEMO, having a higher context size (compared to context size of 1) has a positive effect on prediction accuracy at lower values and a negative effect for very high values (more than 8). The first point can be explained by the fact that increased context size allows our model to identify I/Os more efficiently, as illustrated previously in Figs. 6 and 7. The second point can be explained by the increased context size having a negative impact on learning speed: a higher context size means more nodes in the graph, so building and updating the full model of the application takes more time. The fact that the performance on the other applications, especially LAMMPS, LQCD and Incompact3d, is not affected by the context size suggests that they have simpler I/O call stack patterns. This is confirmed in Table 1, where it can be observed that LAMMPS and LQCD have indeed the highest ratio of directly repeating call stacks. Moreover, it can also be observed that Incompact3d has a number of unique call stack transitions very close to the number of unique call stacks (110 vs 85), which similarly suggests that the application has very simple I/O call stacks patterns. However, this is not a generality: NEMO and NAMD have neither a low unique call stack count (229 for NEMO, 371 for NAMD) nor a too high number of repeating call stacks percentage (32.52% for NEMO, 9.32% for NAMD). In particular, NAMD use Charm++, a message-driven parallel programming framework that makes I/O call stacks more complex and less deterministic. In addition, it has been observed in recent literature that more and more AI-related workloads are executed on HPC platforms, and that those present yet different I/O patterns [29]. In Fig. 13, the size of the model is displayed according to the size of the context. It can be observed that selecting a higher context size makes the model size larger. This is the reason why we chose to use a context size of 2 as the baseline value in this evaluation, in order to maximize prediction accuracy while keeping the model size low.

Effect of the model granularity: In Table 2, the model size is displayed as a function of model granularity for the five applications.

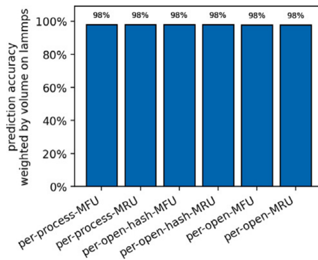

Fig. 19. Effect of model granularity and prediction heuristics on weighted prediction accuracy for LAMMPS.

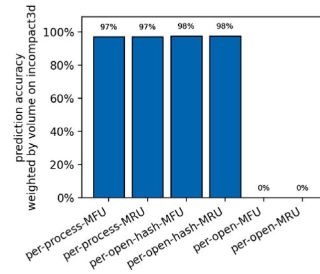

Fig. 22. Effect of model granularity and prediction heuristics on weighted prediction accuracy for Incompact3d.

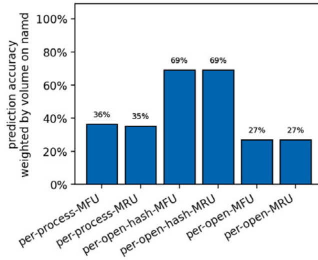

Fig. 20. Effect of model granularity and prediction heuristics on weighted prediction accuracy for NAMD.

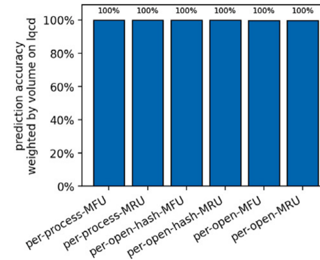

Fig. 23. Effect of model granularity and prediction heuristics on weighted prediction accuracy for LQCD.

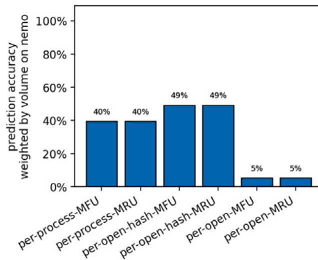

Fig. 21. Effect of model granularity and prediction heuristics on weighted prediction accuracy for NEMO.

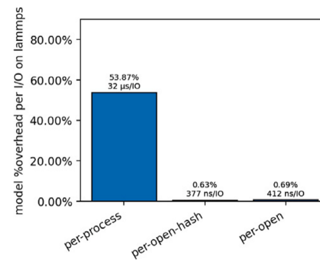

Fig. 24. Effect of model granularity on modeling overheads for LAMMPS.

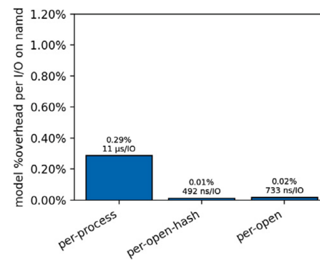

Fig. 25. Effect of model granularity on modeling overheads for NAMD.

All model granularities have similarly small model sizes (less than 20 KB). The per-open-hash granularity is observed to always cause an higher model size than the per-process granularity. This is because over the duration of the application execution, it creates multiples models rather than a single model per process. The per-open granularity, however, is observed to have either the highest model size or the lowest. This is because it creates one model per opened file descriptor, and discards these models as soon as they are closed. As such, the maximum observed model size is high if the application opens a lot of files at the same time, and very low if files are opened sequentially.

In Figs. 14 to 18, the accuracy of the model is displayed as a function of the granularity of the model for different workloads. It can be observed that the per-process granularity has the highest accuracy with this evaluation metric, hence the choice of this heuristic as a baseline configuration. In Figs. 19 to 23, the metric used is the weighted prediction accuracy. The conclusion here is different: the per open call stack hash model granularity has the highest weighted accuracy. This can be explained by this modeling granularity enabling per file recency and frequency metadata, as well as the re-use of models between files with similar open call stacks, which leads to better predictions. The per open modeling granularity similarly enables per file recency and frequency metadata, but it has worse performance on most applications. This is because some of the applications are doing less than a dozen I/Os between each open/close, making the extremely fine-grained modeling with no model re-use a disadvantage as the per open models are discarded before they learned the file I/O behavior.

In Figs. 24 to 28, the prediction and modeling overhead per I/O is displayed as a function of model granularity for all 5 applications. The per process modeling granularity has the highest overhead on

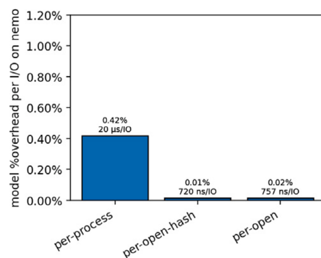

Fig. 26. Effect of model granularity on modeling overheads for NEMO.

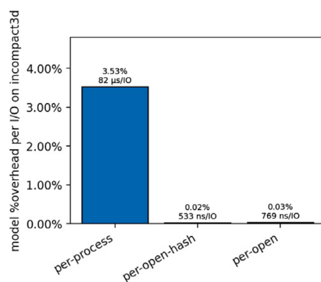

Fig. 27. Effect of model granularity on modeling overheads for Incompact3d.

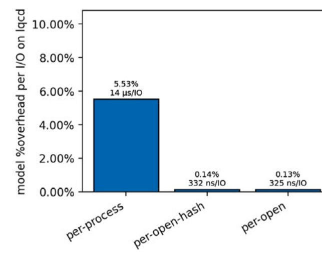

Fig. 28. Effect of model granularity on modeling overheads for LQCD.

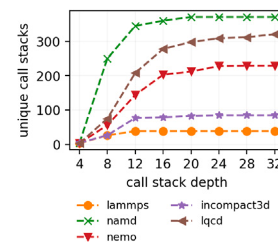

Fig. 29. Effect of call stack depth on call stack differentiation.

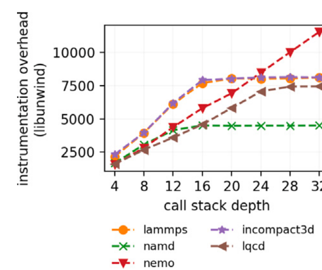

Fig. 30. Effect of call stack depth on *libunwind* instrumentation overhead.

all five applications, with a modeling overhead of up to 82 µs, on Xcompact3d. This high overhead only represents a 3. 5% overhead per I/O, because the average I/O duration of Xcompact3d is relatively high. In contrast, the overhead for the per-process modeling granularity in LAMMPS is only 32 µs but represent a 54% overhead. This very high overhead can be explained by LAMMPS average I/O duration being much lower. Comparatively, the other two modeling granularities present a lower overhead than the per process modeling granularity, with an overhead below 0.1% for the five applications. Furthermore, by integrating the per-I/O overhead data presented in Figs. 24 through 27 with the I/O operation counts listed in Table 1, we can estimate the total I/O overhead incurred by each application. Notably, the maximum observed overhead is 75 s for Incompact3d. The execution of any of the evaluated application taking from 2 h to up to six hours (when configured to have one checkpoint per hour), this overhead can be considered small. Moreover, this overhead is distributed across Incompact3d's deployment on 10 compute nodes and 640 parallel processes, further mitigating its overall impact on performance.

Effect of the prediction heuristic: In Figs. 14 to 23, we can see that the prediction heuristic has an impact on the accuracy of the prediction. Using the MFU heuristic tends to give slightly better prediction accuracy, regardless of the exact accuracy metric chosen. The choice of the prediction heuristic on model size was not evaluated, as both heuristics were supported by all three model implementations. However, the MFU heuristic needs slightly more memory than the MRU heuristic, as MFU needs to store one frequency value per edge, while MRU needs only to store one value per node, with the edge count being most of the time greater or equal to the node count due to repeating call stacks.

7.4.3. Characterizing the call stack instrumentation

As detailed in the background, the call stack instrumentation can be done through the use of the *backtrace* function or through *libunwind*. In both cases, the call stack instrumentation depth is a parameter.

Effect of the call stack instrumentation depth: In Fig. 29, the total number of differentiable call stacks is displayed as a function of call stack instrumentation depth for all five applications. This figure shows that picking lower call stack instrumentation depth means not being able to differentiate every call stack. In Figs. 30 and 31, the call stack instrumentation overhead is displayed as a function of call stack instrumentation depth. It can be observed that an increased call stack instrumentation depth led to an increased instrumentation overhead. Moreover, we observe that the call stack instrumentation overhead closely follows the total number of differentiable call stacks. As such, it seems that it is possible only to decrease the call stack instrumentation overhead by reducing the number of differentiable call stacks, which might possibly undermine the usability of the predicted call stack for prefetch. This is why we choose in this paper to pick a very high call stack instrumentation depth of 32 in order to perfectly differentiate the call stacks.

Comparison of *backtrace* and *libunwind*: In Figs. 30 and 31, the call stack instrumentation overheads of *backtrace* and *libunwind* are compared. We observe that for all but one application, the overhead on *libunwind* is similar but smaller than on *backtrace*, with gains up to 10%. For Incompact3d, however, we observed *libunwind* to have an overhead up to 50% smaller than *backtrace* (2.5 µs vs 5 µs for a call stack depth of 4, 8 µs vs 12.5 µs for a call stack depth of 32). Our hypothesis is that this performance gap is caused by *libunwind* better support of DWARF frame unwinding in the absence of frame pointers [30]. At last, it is also important to note that libunwind is an external, dynamically linked library. Its integration introduces additional complexity for system administrators in terms of installation, maintenance, and compatibility.

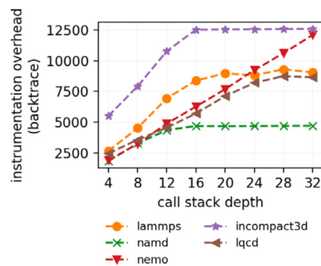

Fig. 31. Effect of call stack depth on *backtrace* instrumentation overhead.

7.5. Performance results summary

- While GrIOt correctly predicts fewer or a similar number of call stacks compared to state-of-the-art Omnisc'IO, the total data volume associated with GrIOt's correct predictions is larger.
- The overhead of GrIOt in terms of model size and time overhead is much lower than in previous work.
- From the 3 granularities evaluated, the per process granularity gave the better accuracy in terms of number of I/Os. The per open call stack hash granularity (model with memory) not only enables the prediction of the next I/O of a given file (as opposed to the next I/O of a given process), but it also gave the best accuracy in terms of I/O volume. The per open granularity (model without memory) did not show good performance.
- Finally, we presented an in-depth study of call stack instrumentation overhead, and most notably have highlighted a trade-off between call stack instrumentation precision and instrumentation overhead while tuning the call stack instrumentation depth.

8. Related work

The most straightforward approach to prefetch data is to modify the application source code, treat it as a white box, and have the user add hints [7] or pre-execute I/Os. These methods, while theoretically viable, can be difficult to apply since the source code of HPC applications is often unavailable or too complex to modify [6].

In Omnisc'IO [17,18] it is observed that most HPC applications do very structured I/Os. As such, the authors used call stack snapshots to represent I/O structures and use them for prediction, making it a gray box approach as it does not need the application source code. At runtime, I/Os are intercepted and a snapshot of their call stack taken, forming a sequence. The structure of this sequence is extracted using a lossless compression algorithm called Sequitur [20]. When an application exhibits very structured and repetitive I/O behavior, this approach is highly efficient. However, when this hypothesis is not true, the size of the model grows very fast, and we observed in our evaluation that the prediction performance suffers great latencies. Also, when modeling irregular applications, the Omnisc'IO prediction algorithm will often have multiple fitting predictors in the grammar. Because Omnisc'IO does not come with heuristics or statistics to choose between these, it will have a hard time making precise long-term predictions. Some later studies [19] built upon Omnisc'IO, but they kept a very similar approach centered around Sequitur and are not applying their models to I/Os.

In our prior work [21], we presented a first implementation of GrIOt that only considered a granularity level per process. In this paper, 2 granularity levels were added to GrIOt, enabling a **better precision**, a **lower overhead**, and the **ability to predict the next I/Os of a given file** rather than being limited to predicting the next I/Os of a process. In addition, all three modeling granularities were **implemented in C**, which enabled a **more realistic evaluation**, and an easier comparison with the state of the art. At last, in this paper we went a step further in the **characterization of call stack instrumentation overhead** compared to the existing literature, bringing to light the relation between call stack instrumentation cost and the number of differentiable call stacks and paving the way for future studies.

Finally, there are several black box methods that do not use call stacks or application source codes to get insight into an application I/O structure. They make predictions from previously traced I/O operations. One such method is pattern matching [13–15,31]. Their main downside is that their overhead grows most notably with the number of detected patterns, and eventually with the size of the time window considered. This limits the number of patterns that can be learned for a given application or file. Another approach is to build probabilistic models [11,12], such as Markov Chains [32]. Most notably with Lynx [12], Markov chains are used to predict the next blocks to be accessed. The model presented suffers a scalability issue as the tracked number of blocks grows. This makes it hard to use for large HPC applications. Some other methods are limited to a specific field. For example, on providing quality prefetch to distributed machine learning [33].

9. Conclusion

This paper presents GrIOt, a full strategy for Graph-based Modeling of HPC Application I/O Call Stacks for Predictive Prefetch. GrIOt makes it possible to use call stacks to predict I/Os for prefetching on both regular and irregular applications with similar or better accuracy compared to previous studies based on grammar representation. Compared to its earlier versions, it allows for a larger configuration space thanks to its new modeling granularities, which most notably enables per-file I/O prediction.

As a perspective for future work, we consider adapting GrIOt implementation to work with MPI-IO. Moreover, because our current implementation creates multiple GrIOt models, we will consider designing a new algorithm to generate a single application model from multiple process models and/or file models. This algorithm would consider MPI rank and apply clustering on the models to identify similar I/O behavior and patterns depending on the MPI rank. We also consider exploring the optimization of the call stack instrumentation overhead, by learning, depending on the current node in the graph, the minimal call stack instrumentation depth keeping all the useful call stack information. A third direction for improvement would be to predict the next I/Os using both the per process and the per open call stack hash modeling granularity, and to select the best model to use depending on the last I/O call stack.

Finally, while the current evaluation highlights the strong potential of GrIOt for I/O prediction, we acknowledge that its practical impact can only be fully assessed through integration into a functional prefetcher. As a next step, we plan to leverage GrIOt's prediction capabilities to design such a prefetcher. The underlying graph structure is well-suited for this purpose, as it can be extended with additional metadata such as return values for prefetching. We thus consider GrIOt a promising foundation for building a learning prefetcher.

CRediT authorship contribution statement

Louis-Marie Nicolas: Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Conceptualization. **Salim Mimouni:** Writing – review & editing, Resources, Methodology, Investigation, Conceptualization. **Philippe Couvée:** Writing – review & editing, Writing – original draft, Supervision, Resources, Project administration, Methodology, Conceptualization. **Jalil Boukhobza:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Louis-Marie Nicolas reports financial support was provided by Atos/Eviden.

Data availability

The code is available on the following link https://github.com/ShinySilver/Graph-Based-IO-Modeling-using-Call-Stacks.

References

[1] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, Zili Shao, Emerging NVM: A survey on architectural integration and research challenges, ACM Trans. Des. Autom. Electron. Syst. 23 (2) (2017) http://dx.doi.org/10.1145/3131848, URL https://doi-org.ins2i.bib.cnrs.fr/10.1145/3131848.

[2] J. Lüttgau, M. Kuhn, K. Duwe, Y. Alforov, T. Ludwig, Survey of storage systems for high-performance computing, Supercomput. Front. Innov. 5 (1) (2018) 31–58, http://dx.doi.org/10.14529/jsfi180103.

[3] Jalil Boukhobza, Pierre Olivier, Flash Memory Integration: Performance and Energy Issues, Elsevier, 2017, https://dl.acm.org/doi/10.1145/3452741.

[4] Jalil Boukhobza, Pierre Olivier, Wen Sheng Lim, Liang-Chi Chen, Yun-Shan Hsieh, Shin-Ting Wu, Chien-Chung Ho, Po-Chun Huang, Yuan-Hao Chang, A survey on flash-memory storage systems: A host-side perspective, ACM Trans. Storage (2025) http://dx.doi.org/10.1145/3723167, in press.

[5] Francieli Zanon Boito, Luan Teylo, Mihail Popov, Théo Jolivel, François Tessier, Jakob Luettgau, Julien Monniot, Ahmad Tarraf, André Carneiro, Carla Osthoff, A deep look into the temporal I/O behavior of HPC applications, 2025, http://dx.doi.org/10.1109/IPDPS64566.2025.00072.

[6] Hariharan Devarajan, Anthony Kougkas, Prajwal Challa, Xian-He Sun, Vidya: Performing Code-Block I/O Characterization for Data Access Optimization, in: 2018 IEEE 25th International Conference on High Performance Computing, HiPC, IEEE, 2018, pp. 255–264, http://dx.doi.org/10.1109/HiPC.2018.00036.

[7] Christina M. Patrick, Mahmut Kandemir, Mustafa Karaköy, Seung Woo Son, Alok Choudhary, Cashing in on hints for better prefetching and caching in PVFS and MPI-IO, in: HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, Association for Computing Machinery, New York, NY, USA, 2010, pp. 191–202, http://dx.doi.org/10.1145/1851476.1851499.

[8] Mohammed Islam Naas, François Trahay, Alexis Colin, Pierre Olivier, Stéphane Rubini, Frank Singhoff, Jalil Boukhobza, EZIOTracer: unifying kernel and user space I/O tracing for data-intensive applications, in: Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems, 2021, pp. 1–11, https://dl.acm.org/doi/10.1145/3439839.3458731.

[9] Chen Wang, Jinghan Sun, Marc Snir, Kathryn Mohror, Elsa Gonsiorowski, Recorder 2.0: Efficient parallel I/O tracing and analysis, in: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, IEEE, 2020, pp. 1–8, http://dx.doi.org/10.1109/IPDPSW50202.2020.00176.

[10] Huong Luu, Babak Behzad, Ruth Aydt, Marianne Winslett, A multi-level approach for understanding I/O activity in HPC applications, in: 2013 IEEE International Conference on Cluster Computing, CLUSTER, IEEE, pp. 23–27, http://dx.doi.org/10.1109/CLUSTER.2013.6702690.

[11] Vivekanand Vellanki, Ann L. Chervenak, A cost-benefit scheme for high performance predictive prefetching, in: SC '99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing, Association for Computing Machinery, New York, NY, USA, 1999, pp. 50–es, http://dx.doi.org/10.1145/331532.331582.

[12] Arezki Laga, Jalil Boukhobza, Michel Koskas, Frank Singhoff, Lynx: a learning linux prefetching mechanism for SSD performance model, in: 2016 5th Non-Volatile Memory Systems and Applications Symposium, NVMSA, IEEE, 2016, pp. 1–6, http://dx.doi.org/10.1109/NVMSA.2016.7547186.

[13] Francieli Zanon Boito, Ramon Nou, Laércio Lima Pilla, Jean Luca Bez, Jean-François Méhaut, Toni Cortes, Philippe O.A. Navaux, On server-side file access pattern matching, in: 2019 International Conference on High Performance Computing & Simulation, HPCS, IEEE, 2019, pp. 217–224, http://dx.doi.org/10.1109/HPCS48598.2019.9188092.

[14] Pradeep Subedi, Philip E. Davis, Manish Parashar, RISE: Reducing I/O contention in staging-based extreme-scale in-situ workflows, in: 2021 IEEE International Conference on Cluster Computing, CLUSTER, IEEE, 2021, pp. 146–156, http://dx.doi.org/10.1109/Cluster48925.2021.00021.

[15] Jun Li, Xiaofei Xu, Zhigang Cai, Jianwei Liao, Kenli Li, Balazs Gerofi, Yutaka Ishikawa, Pattern-based prefetching with adaptive cache management inside of solid-state drives, ACM Trans. Storage 18 (1) (2022) 1–25, http://dx.doi.org/10.1145/3474393.

[16] N. Tran, D.A. Reed, Automatic ARIMA time series modeling for adaptive I/O prefetching, IEEE Trans. Parallel Distrib. Syst. 15 (4) (2004) 362–377, http://dx.doi.org/10.1109/TPDS.2004.1271185.

[17] Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, 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, IEEE, 2014, pp. 623–634, http://dx.doi.org/10.1109/SC.2014.56.

[18] Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, Rob Ross, Using formal grammars to predict I/O behaviors in HPC: The Omnisc'IO approach, IEEE Trans. Parallel Distrib. Syst. 27 (8) (2015) 2435–2449, http://dx.doi.org/10.1109/TPDS.2015.2485980.

[19] Alexis Colin, François Trahay, Denis Conan, PYTHIA: an oracle to guide runtime system decisions, in: 2022 IEEE International Conference on Cluster Computing, CLUSTER, IEEE, 2022, pp. 106–116, http://dx.doi.org/10.1109/CLUSTER51413.2022.00025.

[20] C.G. Nevill-Manning, I.H. Witten, Linear-time, incremental hierarchy inference for compression, in: Proceedings DCC '97. Data Compression Conference, IEEE, 1997, pp. 3–11, http://dx.doi.org/10.1109/DCC.1997.581951.

[21] Louis-Marie Nicolas, Salim Mimouni, Philippe Couvée, Jalil Boukhobza, GrIOt: Graph-based modeling of HPC application I/O call stacks for predictive prefetch, in: SC-W '23: Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, Association for Computing Machinery, New York, NY, USA, 2023, pp. 1195–1201, http://dx.doi.org/10.1145/3624062.3624189.

[22] Hector Marco-Gisbert, Ismael Ripoll Ripoll, Address space layout randomization next generation, Appl. Sci. 9 (14) (2019) 2928, http://dx.doi.org/10.3390/app9142928.

[23] James Done, Christopher A. Davis, Morris Weisman, The next generation of NWP: explicit forecasts of convection using the weather research and forecasting (WRF) model, Atmos. Sci. Lett. 5 (6) (2004) 110–117, http://dx.doi.org/10.1002/asl.72.

[24] MurmurHash2, 2023, GitHub, URL https://github.com/aappleby/smhasher.(Online; Accessed 11 July 2023).

[25] Omniscio, 2023, GitHub URL https://github.com/mdorier/omniscio.(Online; Accessed 11 July 2023).

[26] Aidan P. Thompson, H. Metin Aktulga, Richard Berger, Dan S. Bolintineanu, W. Michael Brown, Paul S. Crozier, Pieter J. in't Veld, Axel Kohlmeyer, Stan G. Moore, Trung Dac Nguyen, et al., LAMMPS-a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales, Comput. Phys. Comm. 271 (2022) 108171.

[27] James C. Phillips, David J. Hardy, Julio D.C. Maia, John E. Stone, João V. Ribeiro, Rafael C. Bernardi, Ronak Buch, Giacomo Fiorin, Jérôme Hénin, Wei Jiang, Ryan McGreevy, Marcelo C.R. Melo, Brian K. Radak, Robert D. Skeel, Abhishek Singharoy, Yi Wang, Benoît Roux, Aleksei Aksimentiev, Zaida Luthey-Schulten, Laxmikant V. Kalé, Klaus Schulten, Christophe Chipot, Emad Tajkhorshid, Scalable molecular dynamics on CPU and GPU architectures with NAMD, J. Chem. Phys. 153 (4) (2020) 044130., http://dx.doi.org/10.1063/5.0014475, arXiv:32752662.

[28] xcompact3d/Incompact3d:, 2024, http://dx.doi.org/10.5281/zenodo.5870206, Zenodo, [Online; Accessed 5 January 2024].

[29] Arnab K. Paul, Ahmad Maroof Karimi, Feiyi Wang, Characterizing machine learning I/O workloads on leadership scale HPC systems, in: 2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS, IEEE, 2021, pp. 1–8, http://dx.doi.org/10.1109/MASCOTS53633.2021.9614303.

[30] Théophile Bastian, Stephen Kell, Francesco Zappa Nardelli, Reliable and fast DWARF-based stack unwinding, Proc. ACM Program. Lang. 3 (OOPSLA) (2019) 1–24, http://dx.doi.org/10.1145/3360572.

[31] Jun He, John Bent, Aaron Torres, Gary Grider, Garth Gibson, Carlos Maltzahn, Xian-He Sun, I/O acceleration with pattern detection, in: HPDC '13: Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing, Association for Computing Machinery, New York, NY, USA, 2013, pp. 25–36, http://dx.doi.org/10.1145/2462902.2462909.

[32] James R. Norris, Markov Chains, (no. 2) Cambridge University Press, 1998.

[33] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, Torsten Hoefler, Clairvoyant prefetching for distributed machine learning I/O, in: SC '21: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1–15, http://dx.doi.org/10.1145/3458817.3476181.

Louis-Marie Nicolas is a PhD student at ENSTA Bretagne Lab-STICC and Atos BDS R&D Data Management in France. His PhD topic covers energy and I/O optimization for HPC. He holds an Engineer's Diploma from the Brest National Engineering School (École Nationale d'Ingénieur de Brest).

Salim Mimouni completed his Ph.D. in 2007, focusing on optical disk recording at the Commissariat à l'Energie Atomique in Grenoble. Holding an Engineer's Diploma from the Institut d'Optique Graduate School (Ecole Supérieure d'Optique, 2004), he has applied his knowledge across various domains, including optical systems, diffraction modeling, and image processing. Transitioning to a role in High-Performance Computing (HPC) as a Data Scientist, he has focused on optimizing algorithms and has contributed to advancements in data management and placement systems, as illustrated by his involvement in several patents.

Philippe Couvée has been working in HPC R&D for more than 20 years. He is leading a team of 17 researchers and engineers, developing products that facilitates data access on large supercomputers. The focus of his team is on data centric solutions that combine cache and acceleration technics with advanced instrumentation and data analytics. He is also teaching computer architecture and system programming at CNAM.

Jalil Boukhobza received the electrical engineering (with Hons.) degree from the Institut Nationale d'Electricite et d'electronique (I.N.E.L.E.C) Boumerdes, Algeria, in 1999, and the MSc and PhD degrees in computer science from the University of Versailles, France, in 2000 and 2004, respectively. He is a professor with the ENSTA-Bretagne, a French State Graduate, PostGraduate and Research Institute. He was a research fellow with the PRiSM Laboratory (University of Versailles) from 2004 to 2006. He was an associate professor with the University Bretagne Occidentale, Brest, France, from 2006 to 2020 and is a member of Lab-STICC. He has also been working with the Technology Research Institute (IRT) bcom since 2013. His main research interests include storage system design, performance evaluation and energy optimization, and operating system design. He works on different application domains such as embedded systems, cloud computing, and database systems.