# Omnisc'IO: a grammar-based approach to spatial and temporal I/O patterns prediction

— **Source link** ↗

Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, Robert Ross

**Institutions:** French Institute for Research in Computer Science and Automation, Argonne National Laboratory

Related papers:

- 24/7 Characterization of petascale I/O workloads

- CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination

- On the role of burst buffers in leadership-class storage systems

- Managing Variability in the IO Performance of Petascale Storage Systems

- Characterization and modeling of PIDX parallel I/O for performance optimization

# Omnisc'IO: A Grammar-Based Approach to Spatial and Temporal I/O Patterns Prediction

Matthieu Dorier,* Shadi Ibrahim,† Gabriel Antoniu,† Rob Ross‡

*ENS Rennes, IRISA, Rennes, France, matthieu.dorier@irisa.fr
†Inria, Centre de Rennes Bretagne-Atlantique, France, {shadi.ibrahim,gabriel.antoniu}@inria.fr
‡Argonne National Laboratory, Argonne, IL 60439, USA, rross@mcs.anl.gov

*Abstract*—The increasing gap between the computation performance of post-petascale machines and the performance of their I/O subsystem has motivated many I/O optimizations including prefetching, caching, and scheduling techniques. In order to further improve these techniques, modeling and predicting spatial and temporal I/O patterns of HPC applications as they run has became crucial.

In this paper we present Omnisc'IO, an approach that builds a grammar-based model of the I/O behavior of HPC applications and uses it to predict when future I/O operations will occur, and where and how much data will be accessed. Omnisc'IO is transparently integrated into the POSIX and MPI I/O stacks and does not require any modification in applications or higher-level I/O libraries. It works without any prior knowledge of the application and converges to accurate predictions within a couple of iterations only. Its implementation is efficient in both computation time and memory footprint.

*Keywords—Exascale, HPC, Storage, I/O, Prediction, Grammar, Omnisc'IO*

## I. INTRODUCTION

Existing petascale computing platforms often fail to meet the I/O performance requirements of applications running at scale. This weakness of the I/O system relative to computing capabilities is part of a trend that is worsening as we develop ever more capable computing platforms, and we expect the next generation of systems to have an even larger gap [1]. Moreover, most HPC applications exhibit a periodic behavior, alternating between computation, communication, and I/O phases. The I/O phases are commonly used for coordinated checkpointing and/or analysis or visualization output. They produce bursts of activity in the underlying storage system [2], [3] that further limit the overall scalability of the HPC application and potentially interfere with other applications running concurrently on the platform [4].

To alleviate the performance penalty of the I/O bottleneck in petascale systems, researchers have explored techniques such as prefetching, caching, and scheduling [4], [5], [6] to improve I/O performance. The effectiveness of such techniques strongly depends on a certain level of knowledge of the I/O access patterns: prefetching and caching indeed require the location of future accesses (i.e., spatial behavior), while I/O scheduling leverages estimations of I/O requests interarrival time (i.e., temporal behavior). The key challenges inherent in these techniques include the proper comprehension and exploitation of the application's I/O behavior within the I/O stack itself [7], [5]. Hence, *modeling and predicting the applications I/O behavior are of utmost importance*.

While predicting the I/O patterns of HPC applications has long been an important goal in large-scale supercomputers, researchers have focused mainly on statistical methods (e.g., hidden Markov models (HMM) [8] and ARIMA models [6]). These approaches often focus exclusively on either spatial or temporal I/O behaviors. Furthermore, they require a large number of observations to accomplish good prediction; hence, they either need long execution time (several runs in some cases) [9] or are doomed to offline trace-based training [10] in order to converge.

Our work addresses the limitations of current prediction systems and takes a step toward intelligent I/O management of HPC applications in next-generation post-petascale supercomputers [11] that is capable of run-time analysis and adaptation to the I/O behavior of applications. To this end, we present the design and implementation of Omnisc'IO, a grammar-based approach for modeling and predicting the I/O behavior of *any* HPC application.

The intuition behind Omnisc'IO is that, on the one hand, while statistical models are appropriate mostly for phenomena that exhibit a random behavior, the (mostly) deterministic behavior of HPC applications, inherent from their code structure, makes other representations of their I/O behavior possible. On the other hand, formal grammars, as natural models to form a sequence of symbols, have been widely applied to areas of text compression, natural language processing, music processing, and macromolecular sequence modeling [12] [13]. Therefore, an approach based on formal grammars is suitable for I/O behavior modeling, since it detects the hierarchical nature of the code that produced the I/O patterns, with its nested loops and stacks of function calls. *To the best of our knowledge, grammar-based models have never been used in the context of HPC applications. Omnisc'IO is the first prediction system that adopts this appealing approach.*

**Contributions:** Omnisc'IO builds a grammar-based model of the I/O behavior of *any* HPC application and uses it to predict *when* future I/O operations will occur (i.e., predict the interarrival time between I/O requests), as well as *where* and *how much* data will be accessed (i.e., predict the file being accessed as well as the location –offset and size– of the data within this file). It learns its model at run time using an algorithm inspired by Sequitur [14]. Sequitur is designed to build a grammar from a sequence of symbols and has been used mainly in the area of text compression; one of the main challenges in our work was to turn it into an algorithm capable of *making predictions*. In this respect, our algorithmic contribution is much more general than its application to I/O pattern prediction.

Omnisc'IO is *transparently* integrated into the POSIX and MPI I/O stacks and *does not require any modification in applications or higher-level I/O libraries*. It works *without any prior knowledge* of the application, and *it converges to accurate predictions within a couple of iterations*. Its implementation is efficient both in computation time (less than a few microseconds to update the model on a recent x86 hardware) and in memory footprint. Omnisc'IO is evaluated with four real HPC applications –CM1, Nek5000, GTC, and LAMMPS– using a variety of I/O backends ranging from simple POSIX to Parallel HDF5 on top of MPI-I/O. Our experiments show that Omnisc'IO achieves from 79.5% to 100% accuracy in spatial prediction (percentage of matching between the predicted segment and the effectively accessed segment) and an average precision of temporal predictions (absolute difference between the predicted and the actual date of the next accesses) ranging from 0.2 seconds to less than a millisecond.

**Goals:** The primary goal for Omnisc'IO, and the focus of this paper, is modeling the I/O behavior of *any* HPC application and using this model to accurately predict the spatial and temporal characteristics of future I/O operations. Omnisc'IO can therefore be applied at the core of many I/O optimizations, including prefetching, caching, or scheduling.

In order not to undermine the generality of our approach, this paper does not present the use of Omnisc'IO in a particular context (i.e., prefetching, caching, or scheduling). Others have already demonstrated the benefits of applying I/O predications to enhance the performance in each of these techniques [8], [6], [9], [10]. We focus our study on the prediction capabilities of Omnisc'IO.

The paper is organized as follows. Section II presents our contribution, followed by an extensive evaluation of its prediction capabilities in Section III. Related work that has attempted spatial and/or temporal I/O access predictions in HPC applications is presented in Section IV. Section V presents our conclusions and suggestions for future work.

## II. OMNISC'IO APPROACH

This section first gives an overview of Omnisc'IO, then dives into its technical and algorithmic details.

### A. Overview of Omnisc'IO

Figure 1 presents an overview of Omnisc'IO. Omnisc'IO captures each atomic request to the file system (open, close, read, write) in a transparent manner, without requiring any change in the application or I/O libraries. At each operation, Omnisc'IO operates as follows.

1) The *context* in which the operation is executed is extracted by recording the call stack of the program (upper-left part of Figure 1(a)). This is a known technique [15] that helps capture the structure of the code that issues the I/O operations. The context is abstracted as a *context symbol* (*a* in the figure).
2) A grammar-based model of the stream of context symbols (upper-right part of Figure 1(a)) is updated by using the Sequitur algorithm. Sequitur has been applied to text compression in the past [12] because of its ability to detect several occurrences of substrings in a text and to store them into grammar rules. We have adapted it to model the repetitive behavior of an HPC application, represented as a stream of
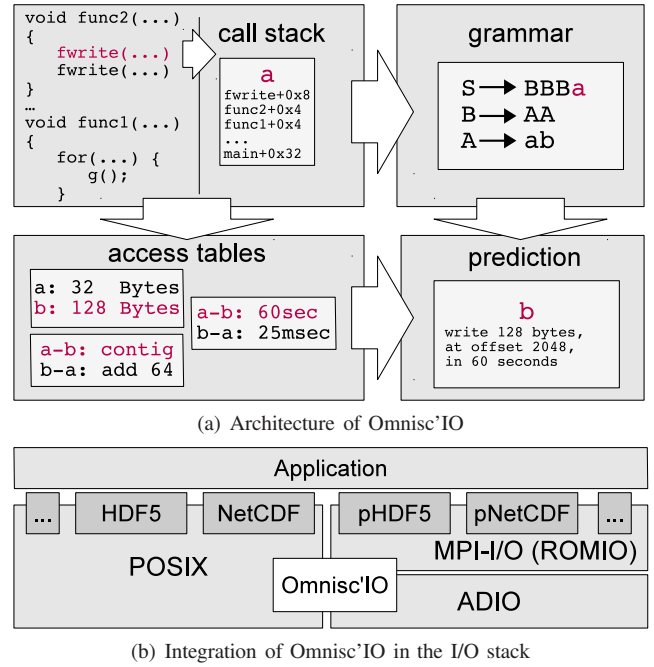


(a) Architecture of Omnisc'IO



(b) Integration of Omnisc'IO in the I/O stack

**Fig. 1:** Overview of the Omnisc'IO approach.

context symbols. The application of Sequitur to the field of application behavior modeling is novel and constitutes part of our contribution.

3) Spatial (size, offset, file) and temporal (interarrival time) access patterns are recorded in tables associating context symbols or transitions among symbols with access patterns (lower-left part of Figure 1(a)). The intuition is (for the example of the access size) that a given context symbol will often be associated with the same access size or with a reduced number of sizes whose sequence can also be learned.

4) We improved on the Sequitur algorithm to *make predictions* of future context symbols. It then becomes easy to predict the characteristics of future accesses by looking up the access patterns associated to the predicted context symbols in the aforementioned tables (lower-right part of Figure 1(a)) Turning Sequitur into a prediction system is a challenge that, to the best of our knowledge, has never been addressed before. The algorithmic details of our prediction model therefore constitutes another important part of our contribution.

### B. Algorithmic and technical description

As shown in Figure 1(b), Omnisc'IO is integrated within the POSIX I/O layer and the ADIO layer in MPI-I/O. In the following, we provide more details on the four steps described above.

*1) Tracking applications behavior:* To give a context to each atomic I/O operation, we use the libc `backtrace` function to retrieve the list of stacked program counters (array of `void*` pointers). When called within a function `f`, this list of addresses characterizes the series of return addresses that leads from `f` back to `main`. Different calls to `f` in distinct places in the program (or libraries) lead to different *call stack traces*. Omnisc'IO calls `backtrace` within wrappers of I/O functions, stores the returned array in a dictionary, and associates it with a unique integer. In the following, such integers are called *context symbols* and represent the context

in which an I/O operation occurs.

Omnisc'IO is based on the observation that **(1)** a particular context is likely to be associated with fixed parameters (e.g., two calls to `write` within the same context are likely to involve the same amount of data); **(2)** transitions between two contexts can also be associated with fixed parameters (tracking the evolution of the offset can be done by tracking transitions between contexts) and with little-varying transition times; and, most important, **(3)** the stream of context symbols is eventually predictable, and a model of it can be built at run time.

In our prototype, we overloaded the POSIX I/O functions (`write`, `read`...) and the libc functions (`fwrite`, `fread`...) using a preloaded shared library. In MPI-I/O we added an intermediate layer within the ADIO layer in ROMIO, a popular implementation of MPI-I/O [16], to track the lowest-level I/O functions that access files metadata (open, close...) and atomic functions that access contiguous blocks of data and are used by more elaborate I/O algorithms.

While working at the lowest level of the I/O stack is necessary to capture the I/O behavior at a fine grain (i.e., a series of atomic requests to the file system), the use of backtraces lets Omnisc'IO have information also on the upper layers that issued the I/O, including I/O middleware, libraries, and the application itself.

*a) Learning the grammar of the application:* While capturing a stream of symbols representing the behavior of the application, we aim to predict the next symbols given past observations. Omnisc'IO models the stream of symbols using a context-free grammar. This grammar is learned at run time using an algorithm inspired by Nevill-Manning and Witten's Sequitur algorithm [14].

As background, a context-free grammar $G$ is a quadruple $(\Sigma, \mathbb{V}, \mathcal{R}, S)$, where $\Sigma$ is a finite set of *terminal* symbols (in our case the symbols defined by the call stack traces); $\mathbb{V}$ is a finite set of *nonterminal* symbols disjoint from $\Sigma$; $\mathcal{R}$ is a finite relation from $\mathbb{V}$ to $(\mathbb{V} \cup \Sigma)^*$, usually written as a set of rules in the form $A \to x_1...x_k$, where $x_i \in (\mathbb{V} \cup \Sigma)$; and $S \in \mathbb{V}$ is a starting symbol. In the following, we call $x_i$ the *nested symbols* of $A$.

Sequitur builds a context-free grammar from a stream of symbols by updating the grammar at each input. It starts with a single rule $S$. At each new input $x$, it appends $x$ to the end of rule $S$ and recursively enforces two constraints:

---

**Digram uniqueness:** Any sequence of two symbols $ab \in (\mathbb{V} \cup \Sigma)^2$ (digram) cannot appear more than once in all rules. If one does, a new rule $R \to ab$ is created, and the constraints are enforced recursively.

---

**Rule utility:** All rules should be instantiated at least twice. If a rule appears only once, its instance is replaced by the content of the rule, the rule is deleted, and the constraints are enforced recursively.

---

Examples of context-free grammars are given in Table I, some of which violate the Sequitur constraints. In the following, the grammar built from the context symbols is called the *main grammar* of Omnisc'IO. Sequitur has a linear worst-case

**TABLE I:** Examples of context-free grammars. Lowercase letters represent terminal symbols, while uppercase letters represent rules and their instances. Example 1 is correct from a Sequitur perspective. Example 2 violates the rule utility (rule $A$ is used only once; it should be deleted an its only instance should be replaced by its content). Example 3 violates the digram uniqueness (digram $ab$ appears twice; a new rule $B \to ab$ can be created to replace it).

| Example 1 | Example 2 | Example 3 |
|---|---|---|
| $S \to abAAe$ | $S \to abAe$ | $S \to ababAAe$ |
| $A \to cd$ | $A \to cd$ | $A \to cd$ |

complexity both in space and in time.

*b) Predictions using the grammar model:* Sequitur builds a grammar from a stream of symbols, but it does not predict the next incoming symbols from past observations. Therefore, we enriched the algorithm to be able to make such a prediction.

This improvement works by marking some of the terminal symbols in the grammar as *predictors*. This *predictor* characteristic is extended to nonterminal symbols by using the following constraints:

---

**Predictor nesting:** A nonterminal can be a predictor only if at least one of its nested symbols is a predictor.

---

**Predictor utility:** If symbol $x$ (terminal or not) is a predictor in rule $Y$, then there exists at least one rule $Z$ such that an instance of $Y$ is a predictor in $Z$.

---

These constraints enforce that **(1)** if the grammar contains at least one predictor, then rule $S$ contains at least one predictor, and **(2)** all the terminal predictors of the grammar can be reached from a predictor in $S$ (proofs of these properties are trivial). The relations linking predictors together form a direct acyclic graph within the tree structure of the grammar. These two structural properties have to be carefully preserved when updating the grammar.

In order to make predictions from the set of predictors, two operations are defined, respectively, to update the set of predictors and to find new ones.

**Updating predictors:** We call *incrementing a predictor* the operation that consists of unmarking a symbol previously marked as a predictor and marking as a predictor the symbol that immediately follows it in the rule where it appears. *Updating predictors* consists of first unmarking all terminal predictors that did not correctly predict the last input, enforcing the predictor's constraints, and then incrementing all remaining terminal predictors. If a predictor is the last symbol of a rule, then nonterminal predictors that reference it are incremented recursively. Examples of this operation are shown in Table II, where predictor symbols are marked in red and underlined.

**Discovering predictors:** If all predictors have been removed because none of them correctly predicted the last input, a new step is necessary to rebuild a set of predictors. This step is completed by navigating through the grammar and

**TABLE II:** Predictors incrementation matching a given input. The predictors are marked in red and underlined. In the first input, $a$ does not match and disappears from the set of predictors, $c$ matches and is incremented to $d$, and $A$ stays a predictor. In the second example, $d$ matches but has no successor in rule $A$; thus $A$ is incremented to $e$ in rule $S$. The resulting models correspond to the grammars before the input is appended and Sequitur's constraints are applied.

| Before Update | Input | After Update |
|---|---|---|
| $S \rightarrow \underline{a}bA\underline{A}e$ | $c$ | $S \rightarrow abA\underline{A}e$ |
| $A \rightarrow \underline{c}d$ | | $A \rightarrow c\underline{d}$ |
| $S \rightarrow abA\underline{A}e$ | $d$ | $S \rightarrow abAA\underline{e}$ |
| $A \rightarrow c\underline{d}$ | | $A \rightarrow cd$ |

**TABLE III:** Discovery of new predictors matching the last input ($b$, appended at the end of rule $S$). The predictors are marked in red and underlined. The symbol $b$ becomes a predictor wherever it appears, and recursively any rule that leads to an occurrence of $b$ becomes a predictor. The predictors are then updated to predict the next expected input (here $c$ or $e$).

| Before Discovery | After Discovery | After Update |
|---|---|---|
| $S \rightarrow abAAeb$ | $S \rightarrow a\underline{b}AA\underline{e}b$ | $S \rightarrow ab\underline{AA}eb$ |
| $A \rightarrow cdb$ | $A \rightarrow cd\underline{b}$ | $A \rightarrow \underline{c}db$ |

setting as predictors all symbols matching with the last symbol of rule $S$ (after insertion of the last input). Parent rules are also set as predictors recursively wherever they appear. Note that the last symbol of rule $S$ may be a nonterminal, which forces new predictors to be searched only within the context of its corresponding rule and thus reduces the number of predictors and narrows down the prediction. An example of this operation is shown in Table III. After the discovery of these new predictors, an *update* is necessary.

The *prediction* of the model corresponds to the set of terminal symbols marked as predictors after inserting a new input, updating the predictors, and enforcing the constraints. Although statistical methods could be used to weight each predicted symbol with a probability of appearance, considering equal probability for all predicted symbols appeared to be sufficient to achieve good results in our experiments.

To implement our algorithm, we reused the simple C++ code provided by the authors of Sequitur.[1] For comparison, the original code has 358 lines, whereas our improved version has 982 (without counting the I/O wrappers and the code related to access tables, which is explained later).

*2) Context-aware access behavior:* The final step in Omnisc'IO is the actual bookkeeping of per-context access behavior. This is done differently for each type of tracked metrics.

*a) Tracking access sizes:* Access sizes are tracked on a per-context symbol basis, so that predicting the next context symbol helps in predicting the next access size. As shown in Section III, most context symbols are always associated with the same access size each time they are encountered in an

execution, making it easy to predict the exact size of the next accesses given a correct prediction of the next context symbols.

For the minority of symbols associated with several access sizes, Omnisc'IO keeps track of all access sizes encountered and builds a grammar from this sequence of sizes. The sizes constitute the terminal symbols of this grammar, which we call a *local size grammar*. The local size grammar associated with a context symbol is updated whenever the context symbol is encountered, and it evolves independently of the *main grammar* and independently of local size grammars attached to other symbols. It can then be used to make predictions of the size.

If the number of different access sizes observed for a given symbol is too large (typically larger than a configurable constant $N$), the local size grammar is replaced by a simple average, minimum value and maximum value that are updated whenever the context symbol is encountered. For our experiments, after analyzing the distributions of different access sizes per symbol, we choose $N = 24$.

More elaborate methods could be implemented to predict the access sizes for context symbols that exhibit apparently random sizes. We show in Section III, however, that the three cases presented above have been sufficient to cover the behavior of all our applications.

*b) Tracking offsets:* Many prefetching systems, including those implemented in the Linux kernel [17], are based on the assumptions of consecutive accesses; that is, the next operation is likely to start from the offset where the previous one ended. As we will show in our experiments, this assumption is held for the POSIX-based applications that we tested, but it fails for applications that use a higher-level library such as HDF5. Indeed such libraries often move the offset pointer backward or forward to write headers, footers, and metadata.

To predict the offset of the next operation, we define the notion of *offset transformation*. An offset transformation can **(1)** leave the offset as it was at the end of the previous operation (*consecutive access transformation*), **(2)** set it to a specific absolute value (*absolute transformation*), or **(3)** set it to a value relative to the offset after the previous operation (*relative transformation*). Since it is not possible at low level to distinguish between absolute and relative transformations, Omnisc'IO uses absolute transformations only for operations that reset the offset to 0 (`open` and `close`). All other nonconsecutive offset transformations are considered relative to the previous offset.

Omnisc'IO associates transitions between context symbols with offset transformations the same way it associates context symbols with access sizes. For instance, if symbol $B$ follows $A$ in the execution and $A$ left the offset at a value from which $B$ starts, then the transition $A \rightarrow B$ is associated with a *sequential transformation*. When a transition encounters different types of offset transformations, Omnisc'IO builds a *local offset grammar* for the transition. Local offset grammars are the counterpart of local size grammars for offset transformations. If the grammar associated with a transition grows too large (more than 24 symbols in our experiments), Omnisc'IO switches back to always predicting a *sequential offset* transformation for this transition of context symbols.

*c) Tracking files pointers:* For the prediction of offsets to work properly, Omnisc'IO needs to know that two

[1]http://www.sequitur.info/sequitur_simple.cc

consecutive operations work on the same file or that an operation works on a new file or a file that has already been accessed earlier. This is particularly important when accesses to multiple files interleave. The prediction of files accessed is done by recording opened file pointers and associating transitions between symbols to changes of file pointers. Since in our experiments the case of interleaved accesses to different files did not appear, we will not study this particular aspect further.

*d) Tracking interarrival times:* To keep track of the time between the end of an operation and the beginning of the next one, Omnisc'IO uses a table associating transitions between context symbols to statistics on the measured time. These statistics include the minimum and maximum observed, the average, and the variance. We prefer these statistics rather than keeping only the average because they represent the minimum required to answer **(1)** whether an operation will immediately follow (maximum, minimum, average, and variance close to 0); **(2)** whether the next operation will follow in a predictable amount of time (maximum, minimum, and average close to each other, small variance); or **(3)** whether the time before the next operation is more unpredictable or depends on parameters that are not captured by our system (large minimum-maximum interval, large variance). Thus, these statistics, while minimal, give us confidence in the predicted interarrival time, which may be important in the context of scheduling, for example.

To quickly react to changes in interarrival times, Omnisc'IO also keeps a *weighed interarrival average time*, updated every time the transition between symbols is encountered by using the following formula,

$$\widehat{T}_{x \to y}^{weighed} \leftarrow \frac{\widehat{T}_{x \to y}^{weighed} + T}{2}, \qquad (1)$$

where $x \to y$ is the observed transition between context symbols $x$ and $y$ and $T$ is the measured interarrival time. This weighed average is more efficient at making predictions of interarrival time, especially in a context where this interarrival time varies a lot between different observations of the same transition.

*3) Overall prediction process and API:* At each operation, Omnisc'IO updates its models (the main grammar and the tables of access sizes, offset transformations and interarrival times). It then updates its predictors and builds the set of possible next context symbols (this set often consists of a single prediction). From these possible next symbols, a set of triplets *(size, offset, date)* is formed that can then be used by scheduling, prefetching, or caching systems. Although this kind of prediction can easily be extended to the series of $N$ next I/O operations, our experiments will focus only on the capability of Omnisc'IO to predict the next one.

To use Omnisc'IO, any software aiming at optimizing I/O simply needs to be linked against the Omnisc'IO library and to call the following function:

```
int omniscio_next(
        omniscio_req** prediction, int* n)
```

This function allocates the `prediction` array and fills it

**TABLE IV:** List of applications used and their I/O backends.

| Application | Field | I/O Method(s) |
|---|---|---|
| CM1 | Climate | HDF5+POSIX HDF5+MPI-I/O HDF5+Gzip |
| GTC | Fusion | POSIX |
| Nek5000 | Fluid Dynamics | POSIX |
| LAMMPS | Molecular Dynamics | POSIX |

with a set of predicted request structures (including the size, offset, and date) representing the expected next I/O accesses.

## III. EVALUATION

In this section, we evaluate Omnisc'IO on real applications. We first assess its capability to predict the next context symbols, and we show how the grammar grows in size as the application continues to run. We then evaluate its performance in predicting the spatial and temporal characteristics of the next operations.

### A. Platform and applications

All our experiments are carried out at the Nancy site of the French Grid'5000 testbed [18]. The applications run on a Linux cluster consisting of 92 Intel Xeon L5420 nodes (8 cores per node, 736 cores in total), using MPICH 3.0.4. The OrangeFS 2.8.7 parallel file system [19] is deployed on a separate set of 12 Intel Xeon X3440 nodes. All nodes, including the file system's, are interconnected through a 20G InfiniBand network.

The list of applications used is presented in Table IV. These applications are real-world codes representative of applications running on current supercomputers. They have been used on NCSI's Kraken[2] and NCSA's Blue Waters[3] for CM1, ORNL's Titan[4] for GTC and LAMMPS, and ANL's Intrepid and Mira[5] for Nek5000.

For the experiments in this paper, the applications are run on 512 cores of Grid5000, except for Nek5000, which runs on 32 cores. These applications are written in Fortran except for LAMMPS (C++). Most of them use a POSIX I/O interface. To show the generality of our approach with respect to higher-level I/O libraries, CM1 [20], [21] uses the HDF5 [22] I/O library over the default (POSIX) I/O driver, as well as the MPI-I/O driver offered by pHDF5, and Gzip compression over the default POSIX driver. CM1 writes one file per process per I/O phase. The domain decomposition in CM1 is such that the amount of data remains the same over time and across processes. The use of compression exemplifies the case of varying data size in a nonvarying domain decomposition. GTC [23] writes one file per node per iteration, but the amount of data varies between files as particles move from one process to another. Like CM1, the domain decomposition in Nek5000 [24] does not vary over time, but the I/O phase is executed only by the rank 0 after a *reduce* phase. LAMMPS [25], [26] also sends data to the rank
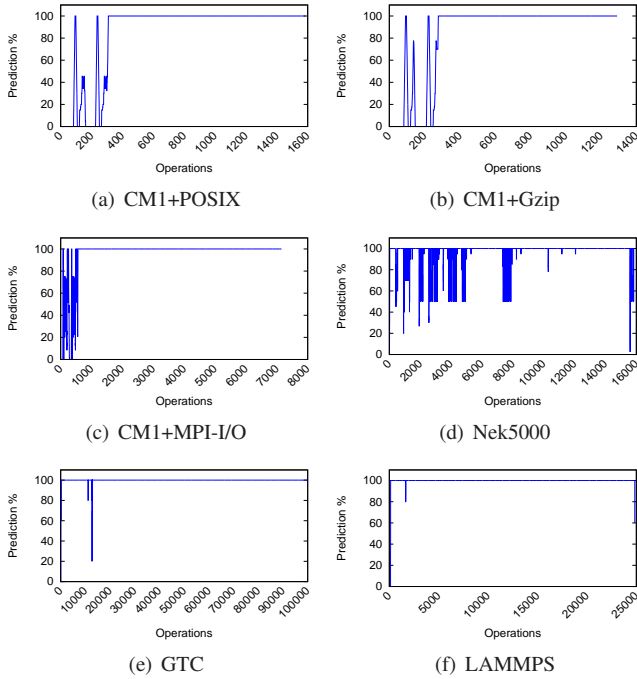
**Fig. 2:** Context prediction capability of Omnisc'IO over the run of each application. Configurations (a), (b), (c), (e), and (f) exhibit a clear learning phase after which Omnisc'IO makes perfect predictions ((e) and (f) exhibit a drop of prediction at the end of the first iteration).

0 process only. This process then writes each set of particles (of potentially different sizes) contiguously in a single file.

Although all processes write data in CM1 and GTC, we consider the results of Omnisc'IO only on process rank 0 (for applications that issue I/O from all processes, these results are in fact identical in all processes since they exhibit the same behavior). We first evaluate how well our algorithm predicts future context symbols based on past observations. We then evaluate the ability of Omnisc'IO to predict the location (offset and size in the file) of the next I/O operations. We also evaluate its ability to predict when future accesses will happen.

### B. Context prediction

To measure context prediction capabilities, we run each application, and at each I/O operation we use Omnisc'IO to predict the context symbol associated with the next one. We use a sliding window of ten operations and report the percentage of correct predictions. When Omnisc'IO predicts several possible next symbols, they are weighed as $\frac{1}{\# \text{ predicted symbols}}$. For instance, if Omnisc'IO predicts that the next symbol will be either $a$ or $b$ and the real next symbol is $b$, then this prediction is weighed $\frac{1}{2}$. For CM1 and GTC, which run for long periods of time, we show only several iterations starting from the beginning of the run.

Results are shown in Figure 2. For all three configurations of CM1 as well as for LAMMPS and GTC, Omnisc'IO converges to a perfect (steady 100%) prediction of symbols after the first iteration. The variation observed during the first iteration corresponds to the moment the grammar starts detecting the innermost loops.

Omnisc'IO seems to learn GTC's behavior (Figure 2(e)) fast: the reason is that GTC's I/O phase consists of a loop over all particles, which is easily modeled in the grammar after the first two particles are written out. The prediction quality drops at the end of the first iteration when Omnisc'IO does not predict the end of this loop and the file being closed. This mistake is never repeated in later iterations. The same pattern appears in LAMMPS.

The case of Nek5000 is more interesting: although it writes periodically the exact same amount of data, the grammar model does not converge as fast and as perfectly as the other applications. Investigating the code of Nek5000, we found that this is due to code branches that process data in a different way depending on its content and then write it in an identical manner, leading to the creation of several symbols that are actually interchangeable in the grammar. Moving the write call outside the branches would help remove this indeterminism. Because we claimed our solution works with no prior knowledge of the application and without the involvement of the application developer, we did not apply this code modification.

We also observe a drop in prediction quality at the end of the LAMMPS and Nek5000 runs. This drop is due to the final results being output in a section of the code different from the one used for the periodic checkpoints. Thus these symbols, which appear the first time at the end of the execution could not have been predicted by any model.

*Cost of a failed prediction:* A failed prediction leads to searching new predictors within the grammar, instead of simply updating existing ones. This operation is linear in the size of the grammar (number of symbols). A failed prediction also has an effect on the system that leverages the prediction. For instance, a prefetching algorithm would read unnecessary data and/or fail to read the data that is actually needed by the program. The real cost would therefore depend on how much the incorrect operation consumes resources that could be used more productively.

### C. Grammar size

Figure 3 shows the evolution of the size of the main grammar as a function of the number of operations. One can clearly distinguish a first *learning phase* during which Omnisc'IO discovers the behavior of the application. This phase corresponds to the first iteration (potentially preceded by an input phase). It is followed by a *stationary regime* during which the model is updated in a mostly logarithmic manner. All the applications considered here exhibit this logarithmic growth of the grammar size after the learning phase. GTC's grammar growth is logarithmic as well, but it exhibits a staircase pattern. This is due to a variable number of particles written at each checkpoint, which leads to a variable number of writes and thus prevents Omnisc'IO from grouping these writes into large rules. That said, after 100,000 accesses the grammar has only 450 symbols.

The memory footprint is directly linked to the size of the main grammar (a symbol in our implementation is a 100-byte C++ object, making the grammar consume 26 KB in the case of CM1+POSIX, for example), and the number of entries in the tables (one entry per symbol or per transition, accounting also for a few bytes. CM1+POSIX uses 198 symbols, for example). This part of the memory footprint does not increases after the
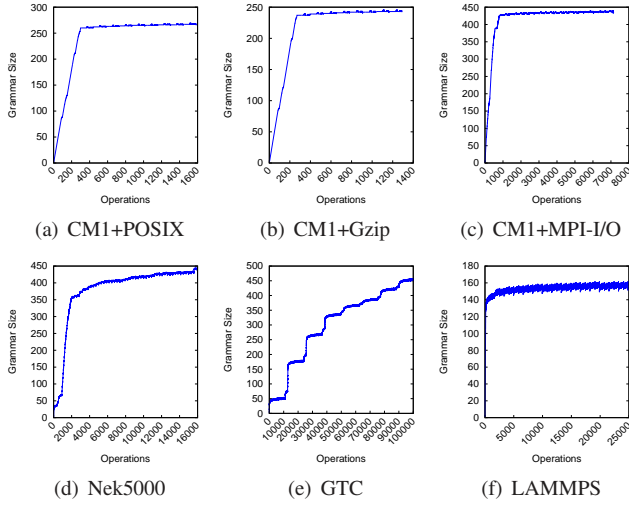
**Fig. 3:** Evolution of main grammar size (sum of the length of each rule, in number of symbols).

learning phase. The memory footprint of Omnisc'IO is thus correlated mainly with the grammar size and does not exceed a few hundreds of kilobytes.

*D. Spatial prediction*

In this section, we first analyze how well our solution predicts the size and the offset of the next operation. We then combine these two predictions to compute a *hit ratio*.

*1) Prediction of sizes:* We analyzed how many different access sizes were associated with each context symbol. We found that the vast majority of symbols were associated with just one size, potentially different for each symbol (171 symbols out of 183 for CM1 using HDF5 are associated with one size, and similar numbers with GZIP and pHDF5, 12 out of 17 for GTC, and all 38 of them for Nek5000). LAMMPS had the most interesting distribution, with 123 symbols associated with a unique size (yet potentially different for each symbol), and one unique symbol associated with a different size at almost every appearance. This distribution is due to the fact that all $n$ processes send their set of particles to the rank 0 process, which writes them into a file in $n$ successive write calls. As the number of particles varies between processes and between checkpoints, this leads to the variation in observed sizes.

To evaluate the prediction of sizes, we use the following relative error as a metric:

$$E_{size} = \frac{|size_p - size_o|}{size_o}, \quad (2)$$

where $size_p$ is the predicted size and $size_o$ is the observed size. Intuitively, if the predictions are always such that $E_{size} \leq \epsilon$, then allocating $1 + \epsilon$ times the predicted size (in a caching system, for example) will always be enough to cover the need for the next operation.

Figure 4 shows the relative error observed for all six cases. In all but Nek5000, the error goes to 0 or close to 0 after the learning phase. Errors observed in Nek5000 match the incorrect predictions of context symbols. In LAMMPS, the prediction is very close but not equal to 0. The reason is that
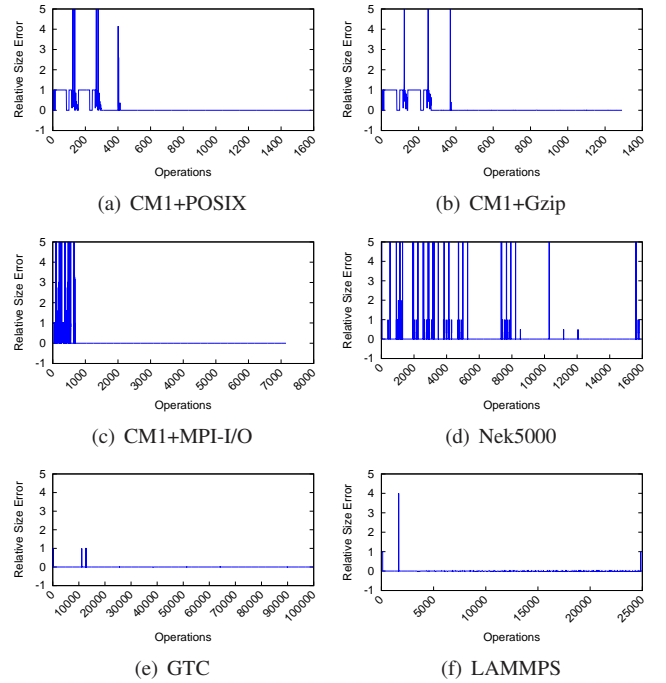


**Fig. 4:** Relative error in the prediction of access sizes in all simulations.

**TABLE V:** Statistics on access sizes by each application.

| Application | Min | Max | Average | Std. dev. |
|---|---|---|---|---|
| CM1 (POSIX) | 4 B | 562.5 KB | 212.9 KB | 268.2 KB |
| CM1 (Gzip) | 4 B | 28.1 KB | 5.5 KB | 10.0 KB |
| CM1 (MPI-I/O) | 4 B | 562.5 KB | 109.5 KB | 219.8 KB |
| Nek5000 | 4 B | 96.0 KB | 20.1 KB | 29.7 KB |
| GTC | 4 B | 8.0 KB | 7.8 KB | 100.0 B |
| LAMMPS | 3 B | 2.70 MB | 851.1 KB | 1.24 MB |

the number of particles written (and thus the size of each write) varies slightly from one write to another. Thus, after trying to build a local size grammar out of those random sizes, Omnisc'IO falls back to keeping track of the average only.

Note that the graphs are cut down to a maximum relative error of 5, whereas the observed errors can be of up to several thousands. For instance, if Omnisc'IO predicts a write or 5,000 bytes while the application actually writes only 2, the relative error is 2,499.

To put these relative errors in perspective, Table V provides statistics on the sizes accessed by each operation. Note that the standard deviation is often close to or even larger than the average access size, making this average a poor estimator if we were to use it in a prediction.

*2) Prediction of offsets:* We consider that an offset prediction is either correct or incorrect. When our algorithm makes several predictions for the next context symbol (and therefore several predictions of offset), correct predictions are weighed accordingly. We compare our solution with the classical *contiguous access* estimation [17], which consists of always predicting that the next offset will follow the previous access. Table VI shows the proportion of contiguous accesses

**TABLE VI:** Proportion of correct offset prediction using a naive *contiguous offsets* approach, and using Omnisc'IO, rounded to closest 0.1%.

| Application | Contiguous Accesses | Omnisc'IO |
|---|---|---|
| CM1 (POSIX) | 47.4% | 92.2% |
| CM1 (Gzip) | 53.2% | 83.0% |
| CM1 (MPI-I/O) | 72.7% | 98.0% |
| Nek5000 | 99.4% | 99.7% |
| GTC | 99.9% | 100% |
| LAMMPS | 99.9% | 100% |

in our set of applications as well as the proportion of correct predictions made by Omnisc'IO. In all cases, Omnisc'IO achieves a better prediction of offsets than does the naive approximation based on contiguous accesses. It is especially better suited when using a high-level I/O library such as HDF5 in CM1, since it manages to model and predict the portion of accesses that are noncontiguous. In particular, the prediction of offset in CM1 using HDF5 goes from 47.4%, when using a contiguous access estimation, to 92.2% with Omnisc'IO.

*3) Hit ratio:* We also combine the prediction of sizes and offsets to measure how accurately our solution can predict the location of the next access. This information forms a predicted segment $S = [\![x_{start}, x_{end}]\!]$. The segment effectively accessed by the next I/O operation is denoted $S_0 = [\![y_{start}, y_{end}]\!]$. The *hit ratio* of $S$ with respect to $S_0$, denoted $H(S|S_0)$, is computed by

$$H(S|S_0) = \begin{cases} \dfrac{100 \times |S \cap S_0|}{\max(x_{end}, y_{end}) - \min(x_{start}, y_{start})} \\ 100 \quad \text{if} \quad S = S_0 = \emptyset \end{cases} \quad (3)$$

This metrics yields the percentage of overlap between the two segments with respect to the distance between their extrema: $H(S|S_0) = 100 \iff S = S_0$. Since our approach may propose several potential next locations, this formula is extended to multiple segments $S_1 \ldots S_n$ by considering the average of $H(S_i|S_0)$ for $i \in [\![1, n]\!]$. Figure 5 shows the results obtained with our simulations, and Table VII presents the average hit ratio over the course of the entire run for each application. Note that for CM1+POSIX and CM1+MPI-I/O, Omnisc'IO holds a perfect hit ratio after the learning phase. Although the hit ratio in LAMMPS also seems to be perfect, it is actually slightly lower than 100% because of the small error made in the prediction of the size (see explanation in Section III-D1). The lowest hit ratio achieved in our experiments was that of CM1+Gzip (79.5%), which, since the study made on the prediction of offsets and sizes in earlier sections, is explained mainly by incorrect predictions of offsets. Our guess is that HDF5 writes compressed data by blocks of predictable size but jumps back and forth in a more unpredictable manner to update metadata.

### E. Temporal prediction

Temporal prediction involves estimating the time between the end of an I/O operation and the beginning of the next one (interarrival time). For qualitative analysis, Figure 6 presents the series of observed interarrival times between consecutive operations, along with the predictions made by Omnisc'IO. We
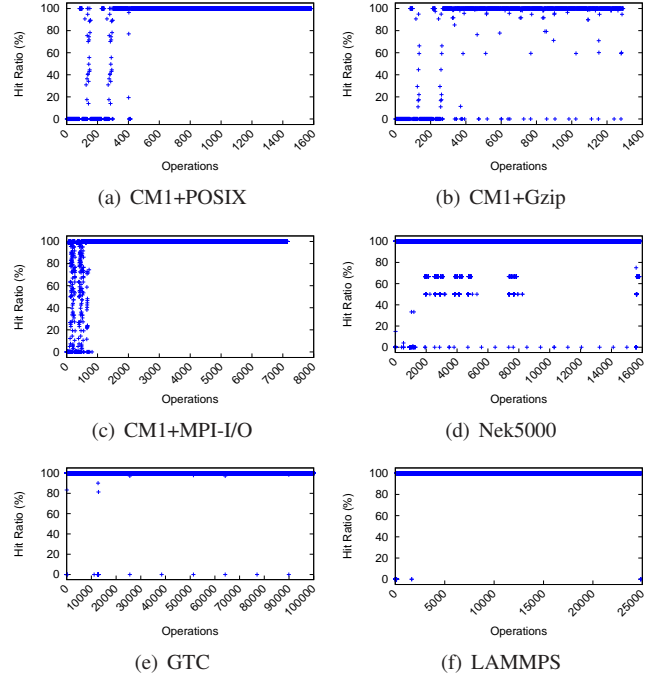


**Fig. 5:** Measurement of the hit ratio using Omnisc'IO to predict the location of the next accessed segment, as a function of the number of the number of operations completed.

**TABLE VII:** Average hit ratio achieved by Omnisc'IO, rounded to closest 0.1%.

| Application | Hit Ratio |
|---|---|
| CM1 (POSIX) | 84.6% |
| CM1 (Gzip) | 79.5% |
| CM1 (MPI-I/O) | 96.0% |
| Nek5000 | 98.6% |
| GTC | 100% |
| LAMMPS | 99.4% |

note that Omnisc'IO is efficient at discriminating *immediate transitions* (low transition times, which can be used as a hint that two consecutive operations belong to the same I/O phase) from *distant transitions* (corresponding to computation and communication phases that last much longer).

Figure 7 presents the absolute difference between observed and predicted transition times on a logarithmic scale. For readability, we consider only the 1,000 last operations of each run, that is, during the stationary regime. Table VIII reports the average of absolute difference over the course of each run (in its entirety, and not restricted to the stationary regime). We also compare the performance of Omnisc'IO with the *immediate reaccess* estimation used by some I/O schedulers (e.g., [27]), which consists of assuming that the next I/O operation is likely to immediately follow the current one (i.e., interarrival time are always estimated to 0) and use a time window during which a potential new operation is expected). In all situations, Omnisc'IO appears to be very good at predicting the interarrival time of I/O accesses. In particular, the average difference between the predicted and observed interarrival time is below a microsecond for LAMMPS, and at
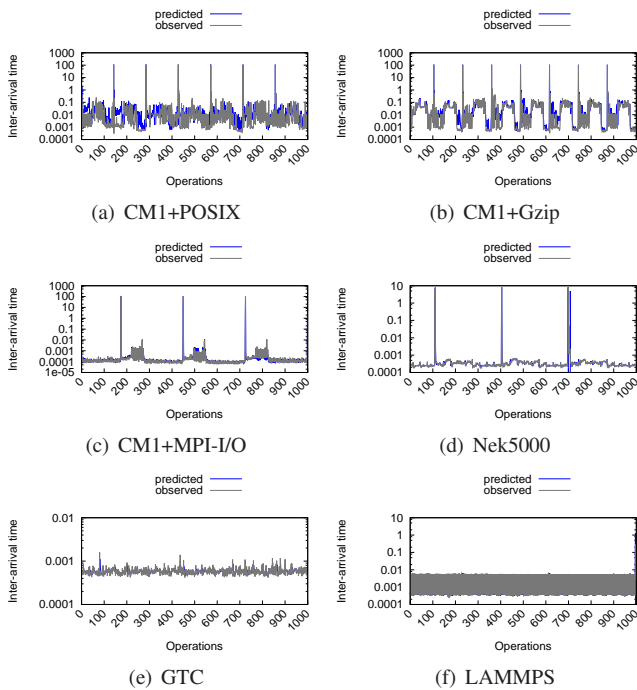
**Fig. 6:** Matching between observed and predicted interarrival times of I/O events.



**Fig. 7:** Difference between predicted and observed interarrival times of I/O events.

**TABLE VIII:** Average time difference between predicted and observed interarrival times (rounded to closest millisecond), and comparison with an *immediate re-access* estimation.

| Application | Time Difference | Immediate Reaccess |
|---|---|---|
| CM1 (POSIX) | 0.197 sec | 0.735 sec |
| CM1 (Gzip) | 0.199 sec | 0.791 sec |
| CM1 (MPI-I/O) | 0.060 sec | 0.406 sec |
| Nek5000 | 0.012 sec | 0.049 sec |
| GTC | 0.001 sec | 0.006 sec |
| LAMMPS | 0.000 sec | 0.003 sec |

worst 0.199 seconds for CM1+Gzip, as opposed to 0.003 and 0.791 seconds, respectively, when considering an immediate reaccess estimation.

Note that by combining the prediction of interarrival times and context symbols, we can estimate how many accesses will happen within a given time window and how many consecutive operations will occur before the end of the I/O phase. Because of space constraints, these studies are not included in this paper.

### F. Limitations of our approach

Like all systems, Omnisc'IO has limitations. As it leans on the repetitiveness of I/O patterns, any nonperiodic applications (e.g., applications that write their results only once at the end of their run) will make Omnisc'IO incapable of discovering repetitive structures in the I/O pattern. To deal with such applications, however, Omnisc'IO can save its model into files and reload it before the next run.

As noted in Section III-B, Omnisc'IO is sensitive to branches in the code that depend on the content of the data. Solving this problem is arguably more difficult, since it would
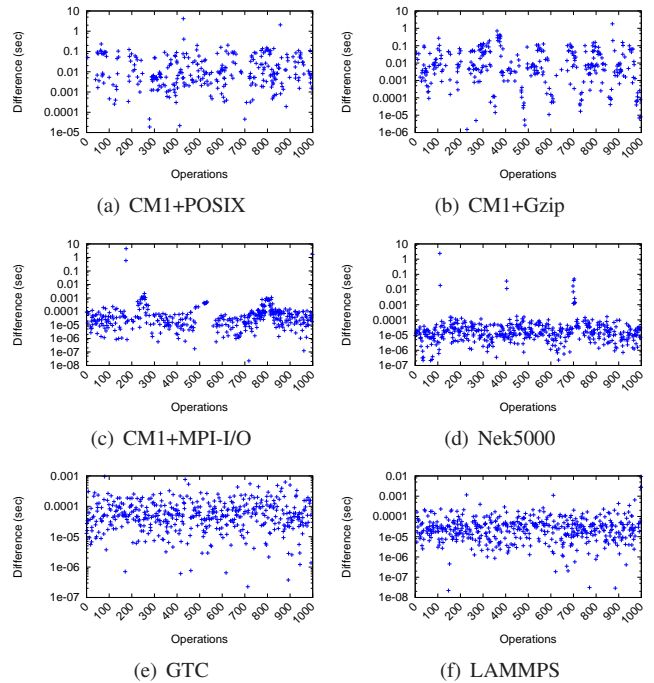
**TABLE IX:** Overhead of Omnisc'IO in the run time of each application (in microseconds per operation).

| Application | Average Overhead | Std. dev. |
|---|---|---|
| CM1 (POSIX) | 20.51 $\mu$sec | 18.27 $\mu$sec |
| CM1 (Gzip) | 20.20 $\mu$sec | 15.56 $\mu$sec |
| CM1 (MPI-I/O) | 19.95 $\mu$sec | 14.50 $\mu$sec |
| Nek5000 | 23.44 $\mu$sec | 18.96 $\mu$sec |
| GTC | 19.03 $\mu$sec | 27.79 $\mu$sec |
| LAMMPS | 22.10 $\mu$sec | 14.72 $\mu$sec |

require Omnisc'IO to know on which specific part of the entire simulation's data the branch depends.

### G. Run-time overhead

The run-time overhead on a commodity hardware is presented in Table IX. This overhead of a few microseconds is negligible compared with the time taken by the I/O operations themselves (a few milliseconds to several seconds). However, since Omnisc'IO works at the level of atomic, contiguous operations, these I/O operations can be made asynchronous to hide the overhead of Omnisc'IO behind the I/O time.

## IV. RELATED WORK

This section presents the related work in the context of grammar-based modeling as well as spatial and temporal I/O prediction.

### A. Grammar-based modeling

The first work related to ours is Sequitur [14]. Sequitur is designed to build a grammar from a sequence of symbols and has been used mainly in the area of text compression [12], but also natural language processing, music processing, and

macromolecular sequence modeling [13]. The repetitive periodic I/O behavior of HPC applications [8] is a very good candidate application for Sequitur. To our knowledge, our approach is the first to take advantage of a grammar-based model not only for modeling but also for making real-time predictions (through improvements of the Sequitur algorithm) of the application's I/O pattern.

### B. I/O patterns prediction

Spacial and temporal I/O access prediction is a challenge commonly addressed in the context of prefetching, caching, and scheduling. Prefetching and caching indeed require a prediction of the location of future accesses [17], while I/O scheduling leverages estimations of I/O requests' interarrival time. Although these domains have been investigated for decades in the context of commodity computers [28], we restrict our study of related works mostly to their use in the HPC area, where applications have different (mostly more regular) I/O behavior.

*1) Spatial predictions:* Most of the work on spatial I/O patterns predication is done to assist I/O prefetching using various approaches, including Markov models [8], speculative execution [29], and knowledge accumulation [5]. These studies predict the I/O behavior based on statistical methods, however, and therefore require either prior knowledge of the application or long runs before the model converges. Moreover, the predictions are evaluated by mean of performance improvements in a particular context such as prefetching. Our work focuses on providing a general approach that can predict both spatial and temporal I/O patterns of any HPC applications, at run time. Its evaluation focuses on its prediction capability, and our results can therefore be transferred to any of the aforementioned applications.

Kroeger and Long [30] study several spatial access pattern modelings techniques, some of which are inspired by text compression algorithms such as variants of PPM (prediction by partial matching). The contexts (or symbols) used in these models are parameters of system-level I/O calls (i.e., file name, offset, size, etc.). Our solution builds a model of the program's structure using backtraces and keeps statistics only on the access parameters. Moreover, it can predict *when* the next operations are going to happen.

Gniady et al. [15] also use stack frames to optimize the prediction of disk accesses, using existing pattern prediction techniques in the operating system. Their solution is used to improve caching.

Madhyastha and Reed [10] use artificial neural networks (ANNs) and hidden Markov models (HMMs) to classify access patterns in order to improve adaptive file systems. In their paper, the authors show that ANNs are incapable of predicting future access patterns, while HMMs need to be trained by using access patterns from several previous executions. The challenge of predicting *when* future accesses will occur is not addressed, however. Our solution based on grammar models is able to converge at run time without prior execution of the application and can predict both spatial and temporal access patterns.

Closer to our approach is the work by He et al. [7], who propose an approach to spatial I/O pattern detection to improve metadata indexing in PLFS. Their approach considers a sequence of *(offset,size)* access parameters and tries to find repetitive patterns in the differences (delta) between consecutive accesses, using a method inspired by the LZ77 sliding window algorithm. They also apply their algorithm to pattern-aware prefetching. While Omnisc'IO targets the same goal, it differs in the underlying algorithm used (Sequitur-inspired versus LZ77-inspired). Our approach also leverages stack traces to build a model of the program's behavior, whereas the solution proposed by He et al. works on the sequence of *(offset,size)* pairs.

*2) Temporal prediction and scheduling:* Prediction of temporal access pattern has been investigated by Tran and Reed [6] using ARIMA time series to model interarrival time between I/O requests. While the authors propose a solution that builds the model at run time, such statistical models need a large number of observations in order to converge to a good representation and, thus, good predictions. While ARIMA-based methods are effective at file system level when no knowledge can be retrieved from the application, we have shown that accurate predictions of interarrival times are possible at the application level without the need for such stochastic methods.

Byna et al. [9] propose a notation called I/O signatures to assist I/O prefetching. I/O signatures describe the historic access pattern including the spatiality, request size, repetitive behavior, temporal intervals, and type of I/O operation. I/O signatures are stored persistently and can be used only in later runs.

Zhang et al. [31] couple I/O schedulers with process schedulers on compute nodes. When an application enters an I/O phase, it spawns new processes that pre-execute the code in order to find future I/O accesses while the main processes are waiting for the first access to complete. The knowledge of future accesses is then leveraged by the main processes. Considering the trend toward smaller operating systems with only restricted features, this kind of approach is likely not to be applicable in future machines with no preemptive process scheduler.

Several schedulers have been proposed that leverage some knowledge from the applications. The network request scheduler from Qian et al. [32], built in Lustre [33], associates deadlines to requests. A similar design is proposed by Song et al. [34]. These schedulers are not based on any prediction, however, and could be greatly improved by knowledge extracted by Omnisc'IO on future access patterns. This knowledge can indeed help decide which application should be given priority to access the file system given its future access pattern. The scheduler proposed by Lebre et al. [35] aims at aggregating and reordering requests while trying to maintain fairness across applications, a task that would undoubtedly be easier with any kind of prediction of future incoming I/O requests.

In our previous work [4] we advocated for cross-application coordination to mitigate I/O interference. While the application user was required to explicitly instrument an application to expose its I/O patterns to other applications, the spatial and temporal I/O predictions presented in the present work can be leveraged to remove the need for this instrumentation and thus offer transparent cross-application I/O scheduling.

## V. Conclusion

The unprecedented scale of tomorrow's supercomputers forces researchers to consider new approaches to data management. In particular, self-adaptive and intelligent I/O systems that are capable of runtime analysis, modeling, and prediction of applications I/O behavior with little overhead and memory footprint will be of utmost importance to optimize prefetching, caching, or scheduling techniques.

In this paper we have presented Omnisc'IO, an approach that builds a model of I/O behavior using formal grammars. Omnisc'IO is transparent to the application, has negligible overhead in time and memory, and converges at run time without prior knowledge of the application.

We have evaluated Omnisc'IO with four real applications in a total of six scenarios. Omnisc'IO converges quickly to a stable model capable of predicting both the date and location of future I/O accesses, achieving a near-perfect hit ratio (from 79.5% to 100% in our experiments) and interaccess time estimation (up to 0.199 sec of average absolute difference with the observed interaccess time).

As future work, we plan to integrate Omnisc'IO within our previous CALCioM framework [4] for efficient I/O scheduling and to implement prefetching and caching systems that leverage the excellent prediction capabilities shown by Omnisc'IO. We also plan to explore this approach as a mechanism for representing I/O behavior for replay in parallel discrete event simulations of large-scale HPC storage systems.

For reproducibility and comparison with later works, traces of all our experiments are available along with the code of Omnisc'IO at http://omniscio.gforge.inria.fr.

## References

[1] A. Geist and R. Lucas, "Major Computer Science Challenges at Exascale," *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 427–436, 2009.

[2] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the Role of Burst Buffers in Leadership-Class Storage Systems," in *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), 2012*. IEEE, 2012, pp. 1–11.

[3] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, ser. FAST'14. Berkeley, CA: USENIX Association, 2014, pp. 213–228.

[4] 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 - International Parallel and Distributed Processing Symposium*, Phoenix, AZ, May 2014.

[5] J. He, X.-H. Sun, and R. Thakur, "KNOWAC: I/O Prefetch via Accumulated Knowledge," in *Proceedings of the 2012 IEEE International Conference on Cluster Computing*, ser. CLUSTER '12. Washington, DC: IEEE Computer Society, 2012, pp. 429–437.

[6] N. Tran and D. A. Reed, "Automatic ARIMA Time Series Modeling for Adaptive I/O Prefetching," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 4, pp. 362–377, Apr. 2004.

[7] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun, "I/O Acceleration with Pattern Detection," in *Proceedings of the 22nd international symposium on High-Performance Parallel and Distributed Computing*. ACM, 2013, pp. 25–36.

[8] J. Oly and D. A. Reed, "Markov Model Prediction of I/O Requests for Scientific Applications," in *Proceedings of the 16th international conference on Supercomputing*. ACM, 2002, pp. 147–155.

[9] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, "Parallel I/O Prefetching using MPI File Caching and I/O Signatures," in *International Conference for High Performance Computing, Networking, Storage and Analysis. SC 2008*, Nov. 2008, pp. 1–12.

[10] T. Madhyastha and D. Reed, "Learning to Classify Parallel Input/Output Access Patterns," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 8, pp. 802–813, Aug. 2002.

[11] F. Isaila, J. Garcia, J. Carretero, R. B. Ross, and D. Kimpe, "Making the Case for Reforming the I/O Software Stack of Extreme-Scale Systems," *Preprint ANL/MCS-P5103-0314, Argonne National Laboratory*, 2014.

[12] J. Kieffer and E.-H. Yang, "Grammar-Based Codes: A New Class of Universal Lossless Source Codes," *IEEE Transactions on Information Theory*, vol. 46, no. 3, pp. 737–754, May 2000.

[13] C. G. Nevill-Manning, "Inferring Sequential Structure," Ph.D. dissertation, University of Waikato, 1996.

[14] C. G. Nevill-Manning and I. H. Witten, "Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm," *Journal of Artificial Intelligence Research*, vol. 7, no. 1, pp. 67–82, Sep. 1997.

[15] C. Gniady, A. R. Butt, and Y. C. Hu, "Program-Counter-Based Pattern Classification in Buffer Caching," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA: USENIX Association, 2004, pp. 27–27.

[16] R. Thakur, W. Gropp, and E. Lusk, "On Implementing MPI-IO Portably and with High Performance," in *Proceedings of the sixth workshop on I/O in Parallel and Distributed Systems*. ACM, 1999, pp. 23–32.

[17] F. Wu, "Sequential File Prefetching in Linux," in *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*. IGI Global, 2010, pp. 217–236.

[18] INRIA, "Aladdin Grid'5000: http://www.grid5000.fr."

[19] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur, "PVFS: a Parallel File System for Linux Clusters," in *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*. Berkeley, CA: USENIX Association, 2000.

[20] Georges Bryan, UCAR, "CM1 http://www.mmm.ucar.edu/people/bryan/cm1/."

[21] G. H. Bryan and J. M. Fritsch, "A Benchmark Simulation for Moist Nonhydrostatic Numerical Models," *Monthly Weather Review*, vol. 130, no. 12, pp. 2917–2928, 2002.

[22] NCSA, "HDF5: http://www.hdfgroup.org/HDF5/."

[23] Plasma Theory Group, UCIrvine, "GTC version 1 http://phoenix.ps.uci.edu/GTC/."

[24] P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier, "Nek5000 http://nek5000.mcs.anl.gov," 2008.

[25] Sandia National Laboratory, "LAMMPS http://lammps.sandia.gov/."

[26] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1 – 19, 1995.

[27] X. Zhang, K. Davis, and S. Jiang, "IOrchestrator: Improving the Performance of Multi-Node I/O Systems via Inter-Server Coordination," in *2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2010, pp. 1–11.

[28] H.-Y. Li, C. S. Xie, and Y. Liu, "A New Method of Pefetching I/O Requests," in *International Conference on Networking, Architecture, and Storage, NAS 2007*, July 2007, pp. 217–224.

[29] Y. Chen, S. Byna, X.-H. Sun, R. Thakur, and W. Gropp, "Exploring Parallel I/O Concurrency with Speculative Prefetching," in *37th International Conference on Parallel Processing. ICPP'08.* IEEE, 2008, pp. 422–429.

[30] T. Kroeger and D. Long, "The Case for Efficient File Access Pattern Modeling," in *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*, 1999, pp. 14–19.

[31] X. Zhang, K. Davis, and S. Jiang, "Opportunistic Data-driven Execution of Parallel Programs for Efficient I/O Services," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International.* IEEE, 2012, pp. 330–341.

[32] Y. Qian, E. Barton, T. Wang, N. Puntambekar, and A. Dilger, "A Novel Network Request Scheduler for a Large Scale Storage System," *Computer Science - Research and Development*, vol. 23, pp. 143–148, 2009.

[33] S. Donovan, G. Huizenga, A. J. Hutton, C. C. Ross, M. K. Petersen, and P. Schwan, "Lustre: Building a File System for 1000-Node Clusters," Citeseer, 2003.

[34] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "Server-Side I/O Coordination for Parallel File Systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY: ACM, 2011, pp. 17:1–17:11.

[35] A. Lebre, G. Huard, Y. Denneulin, and P. Sowa, "I/O Scheduling Service for Multi-Application Clusters," in *Proceedings of IEEE Cluster 2006*, 2006.