Using Formal Grammars to Predict I/O Behaviors in HPC: The Omnisc'IO Approach

M. Dorier, S. Ibrahim, G. Antoniu and R. Ross, in *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2435-2449, Aug. 1 2016.
doi: 10.1109/TPDS.2015.2485980

BLG546E Term Paper Presentation

Tuğrul Yatağan
504161551

Oğuzhan Demir
504161545

May 16, 2018

Outline

Introduction

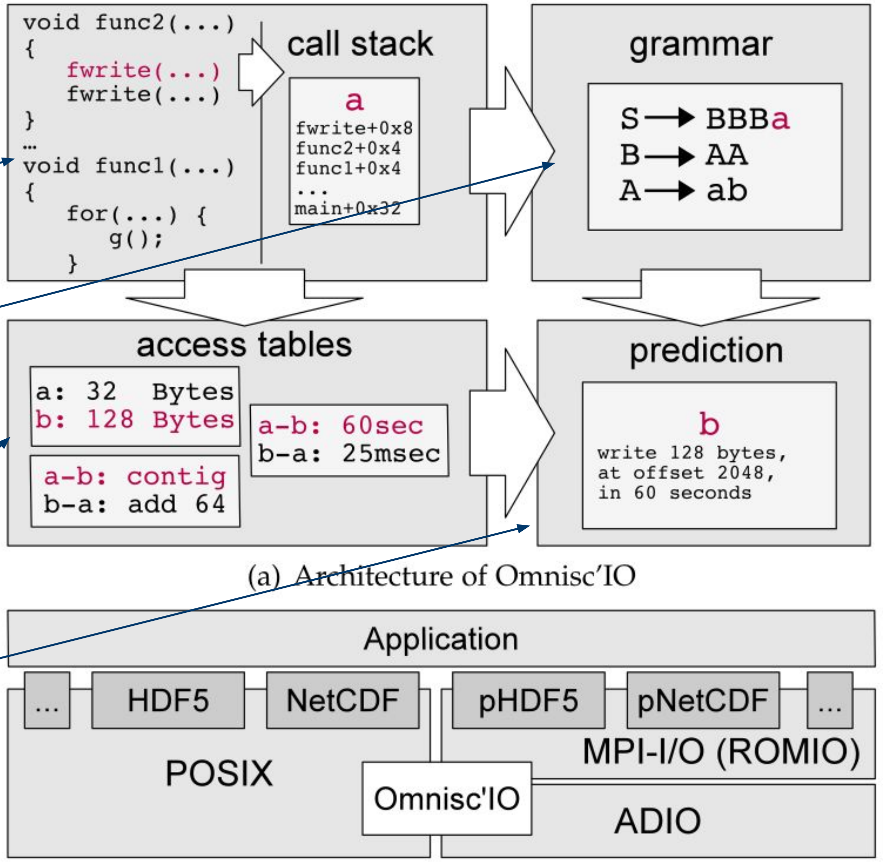
- HPC applications exhibit periodic behavior, alternating between; computation, communication and I/O phases.
  - I/O phases produce bursts of activity in the underlying storage system.
- I/O optimizations;
  - Prefetching, caching, and scheduling.
  - Modeling and predicting spatial and temporal I/O patterns is crucial.
- Omnisc'IO;
  - Approach that builds a grammar-based model of the I/O behavior.
  - Predicts;
    - When future I/O operations will occur?
      - Interarrival time between I/O requests.
    - Where data will be accessed?
      - File being accessed as well as the location.
    - How much data will be accessed?
      - Offset and size of the data within a file.
  - Transparently integrated into the POSIX, does not require any modification in applications or higher-level I/O libraries.
  - Works without any prior knowledge of the application.
  - Converges to accurate predictions of any N future I/O operations.

Introduction

- I/O phases are commonly used for coordinated checkpointing and/or analysis or visualization output.
- Effectiveness of prefetching, caching, and scheduling;
  - Strongly depends on a certain level of knowledge of the I/O access patterns.
  - Require the location of future accesses (spatial behavior).
  - Require estimations of I/O requests interarrival time (temporal behavior).
- Statistical methods and non-statistical methods based on frequent patterns require a large number of observations to accomplish good prediction.
  - Statistical models are appropriate mostly for phenomena that exhibit a random behavior.
- HPC applications exhibit more deterministic behavior since;
  - Absence of interactivity with an end-user.
  - Their code structure.
- Formal grammars (models to represent a sequence of symbols) have been widely applied to; text compression, natural language processing, music processing etc.
- Formal grammars is also suitable for HPC I/O behavior modeling;
  - It detects the hierarchical and periodic nature of the code that produced the I/O patterns.
    - Nested loops and stacks of function calls.

The Omnisc'IO Approach

- Omnisc’IO captures each atomic request to the file system (open, close, read, write, etc.) in a transparent manner, without requiring any change in the application or I/O libraries.
- The context is extracted by recording the call stack of the program.
- A grammar-based model of the stream of context symbols is updated by using StarSequitur a new algorithm derived from Sequitur.
  - Sequitur has been applied to text compression in the past because of its ability to detect several occurrences of substrings in a text and to store them into grammar rules.
- Spatial (size, offset, file) and temporal (interarrival time) access patterns are recorded in tables associating context symbols.
- Makes predictions of future context symbols and reduce the grammar in case of periodic behaviors.



```
void func2(...)
{
  fwrite(...)
  fwrite(...)
}
...
void func1(...)
{
  for(...) {
     g();
  }
```

(a) Architecture of Omnisc’IO

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

Fig. 1. Overview of the Omnisc’IO approach.

Algorithmic and Technical Description

- Tracking Applications Behavior
  - POSIX I/O functions (write, read...) and the libc functions (fwrite, fread...) are overloaded using a preloaded shared library.
  - Backtrace retrieves the list of stacked program counters.
  - Associate the returned array with a unique integer.
  - Same stack trace will be associated with the same integer called context symbols. Stream of symbols represents the behavior of the application
- Sequitur builds a context-free grammar at run time from a stream of symbols by updating the grammar at each input.
- The Sequitur algorithm constructs a grammar by substituting repeating phrases in the given sequence with new rules and produces a concise representation of the sequence. Example;
  S→abcab
  the algorithm will produce;
  S→AcA, A→ab

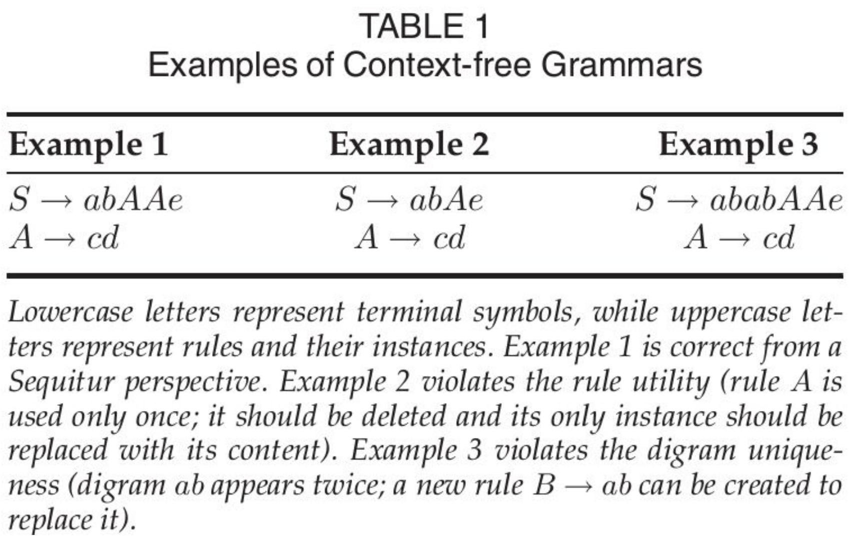


TABLE 1
Examples of Context-free Grammars

Example 1	Example 2	Example 3
$S \rightarrow abAAe$	$S \rightarrow abAe$	$S \rightarrow ababAAe$
$A \rightarrow cd$	$A \rightarrow cd$	$A \rightarrow cd$

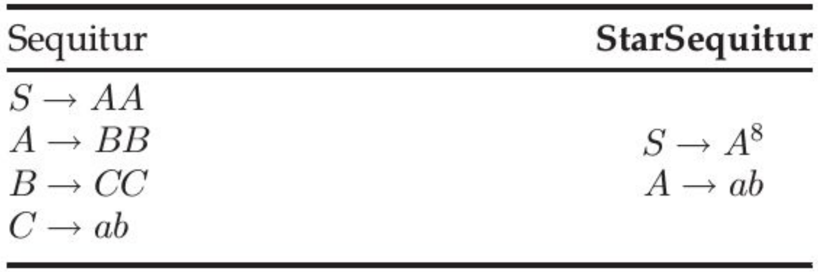
Lowercase letters represent terminal symbols, while uppercase letters represent rules and their instances. Example 1 is correct from a Sequitur perspective. Example 2 violates the rule utility (rule A is used only once; it should be deleted and its only instance should be replaced with its content). Example 3 violates the digram uniqueness (digram ab appears twice; a new rule $B \rightarrow ab$ can be created to replace it).

- Diagram uniqueness: Any sequence of two symbols cannot appear more than once.
- Rule utility: All rules should be instantiated at least twice.

Algorithmic and Technical Description

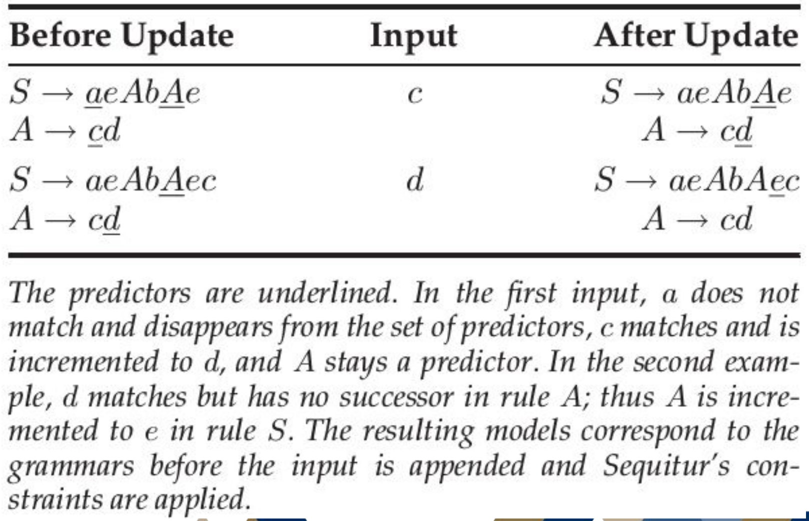
- Most HPC application have a periodic behavior.
  - A new algorithm StarSequitur stores symbols with an exponent, for instance a^{3} instead of aaa.
- Mark some of the terminal symbols in the grammar as predictors.
  - A predictor in the grammar represents a position in grammar corresponding to a pattern that Omnisc'IO "thinks" we are currently encountering again.
  - Updating and discovering predictors are key steps.
- For Omnisc'IO to be useful in HPC I/O optimizations, it should be able to predict any N future operations.
  - Iterators that start at current predictors and read the grammar from them.
- At each operation;
  - Omnisc'IO updates its main grammar, tables of access sizes, offset transformations, interarrival times.
  - Updates its predictors and builds the set of possible next context symbols.
- From these possible next symbols;
  - Scheduling, prefetching or caching can be improved.

TABLE 2
Grammars Built by Sequitur and StarSequitur from the Sequence *abababababababab*



Sequitur	StarSequitur
$S \rightarrow AA$ $A \rightarrow BB$ $B \rightarrow CC$ $C \rightarrow ab$	$S \rightarrow A^{8}$ $A \rightarrow ab$

TABLE 3
Predictors Incrementation Matching a given Input



Before Update	Input	After Update
$S \rightarrow \underline{a}eAb\underline{A}e$ $A \rightarrow \underline{c}d$	c	$S \rightarrow aeAb\underline{A}e$ $A \rightarrow c\underline{d}$
$S \rightarrow aeAb\underline{A}ec$ $A \rightarrow c\underline{d}$	d	$S \rightarrow aeAbA\underline{e}c$ $A \rightarrow cd$

The predictors are 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.

Algorithmic and Technical Description

- Context-Aware I/O Behavior.
  - Tracking Access Sizes
    - Predicting the next context symbol helps in predicting the next access size.
    - 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 local size grammar.
    - Local size grammar associated with a context symbol is updated whenever the context symbol is encountered
  - Tracking Offsets
    - The next operation is likely to start from the offset where the previous one ended.
      - Fails for applications that use a higher-level library that moves the offset pointer backward or forward to write headers, footers, and metadata.
    - Offset transformation needed.
  - Tracking Files Pointers
    - The prediction of files accessed is done by recording opened file pointers and associating transitions between symbols.
  - Tracking Interarrival Times.
    - Keep track of the time between the end of an operation and the beginning of the next one.

Evaluation

- List of applications to be used while experimenting Omnisc'IO
- These applications are real world examples. They are all built for specific purpose.
- Test Environment
  - Nancy site of the French Grid5000
  - Linux cluster Intel Xeon L5420 nodes (512 cores)
  - The OrangeFS 2.8.7 parallel file system is deployed on a separate set of 12 Intel Xeon X3440 nodes
  - 20 G InfiniBand network

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

Evaluation

Immediate Context Prediction

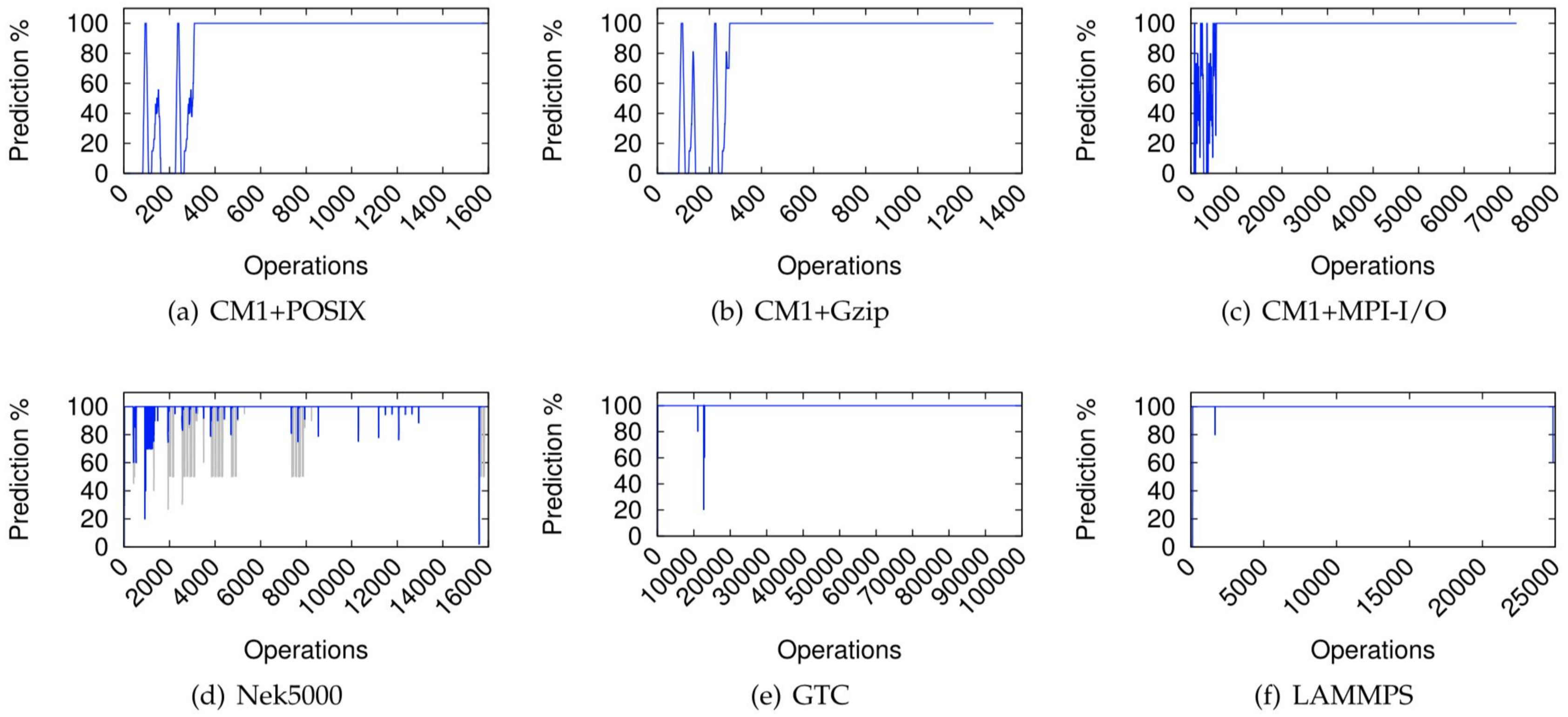
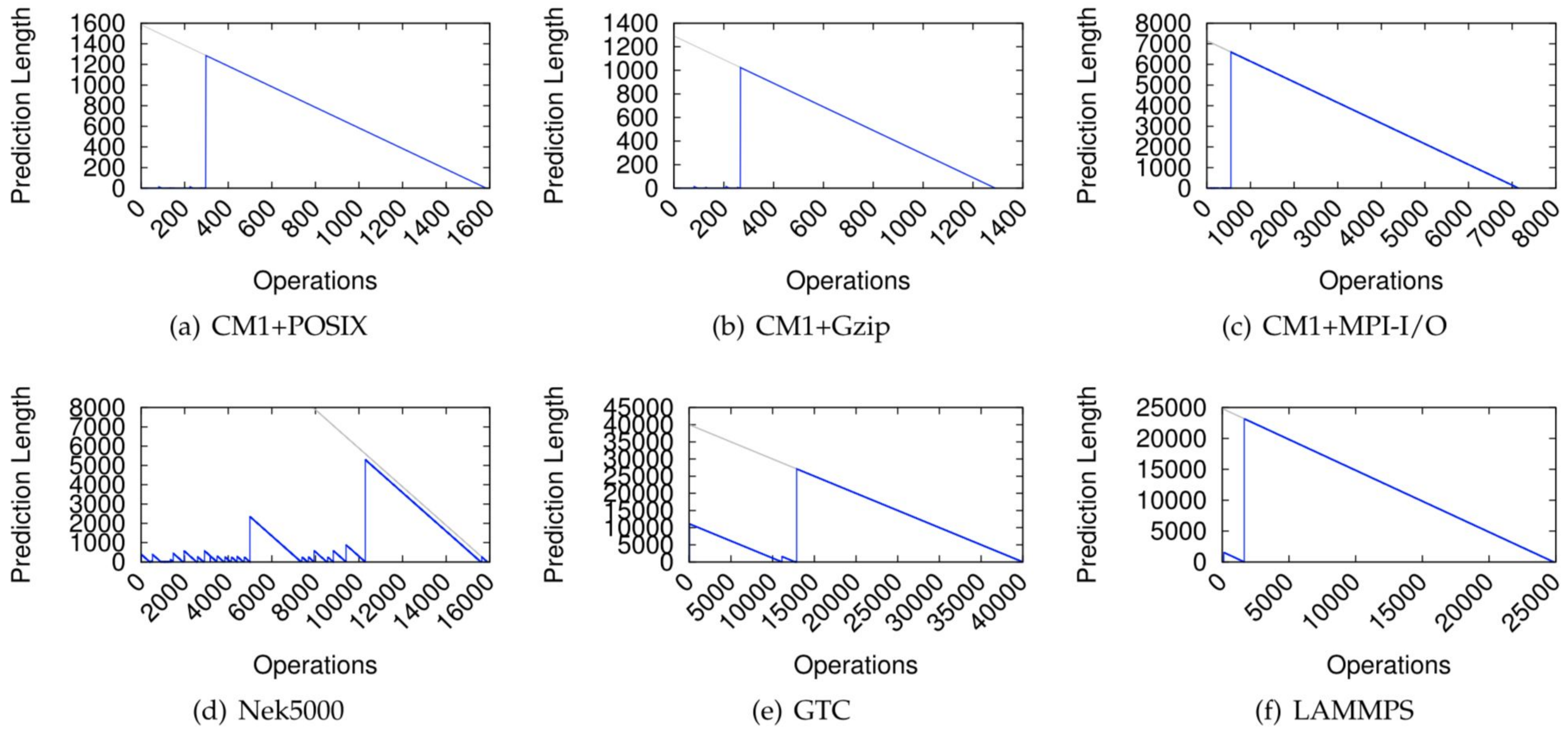


Fig. 2. Context prediction capability of Omnisc'IO for the next context symbol only, over the run of each application. Configurations (a), (b), (c), (e), and (f) exhibit a clear learning phase after which Omnisc'IO makes perfect predictions ((e) and (f) exhibit a drop of prediction at the end of the first iteration). In (d) the gray curve corresponds to the results from our previous work, without the weighting system.

Prediction of N Next Symbols



(a) CM1+POSIX (b) CM1+Gzip (c) CM1+MPI-I/O
(d) Nek5000 (e) GTC (f) LAMMPS

Fig. 3. Number of symbols that Omnisc'IO correctly predicts over the course of each application's run. This number is bounded by the remaining number of I/O operations (materialized by the grey line). Given the computation cost of performing predictions up to the end of the simulation for each I/O operations, we reduced the length of GTC's run in this experiments to 40,000 operations (three I/O phases) rather than 100,000 previously.

Evaluation

Grammer Size

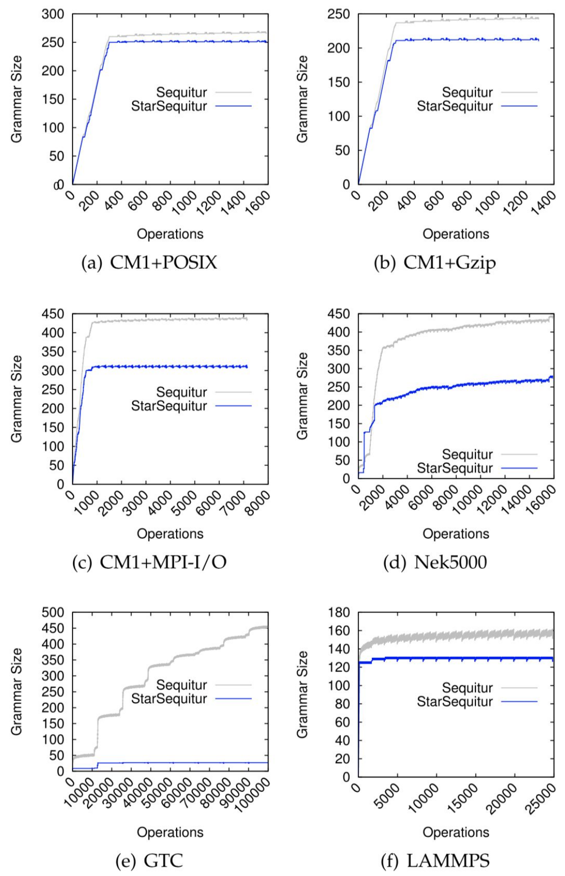


Fig. 4. Evolution of main grammar size (sum of the length of each rule, in number of symbols).

Evaluation

Relative Error

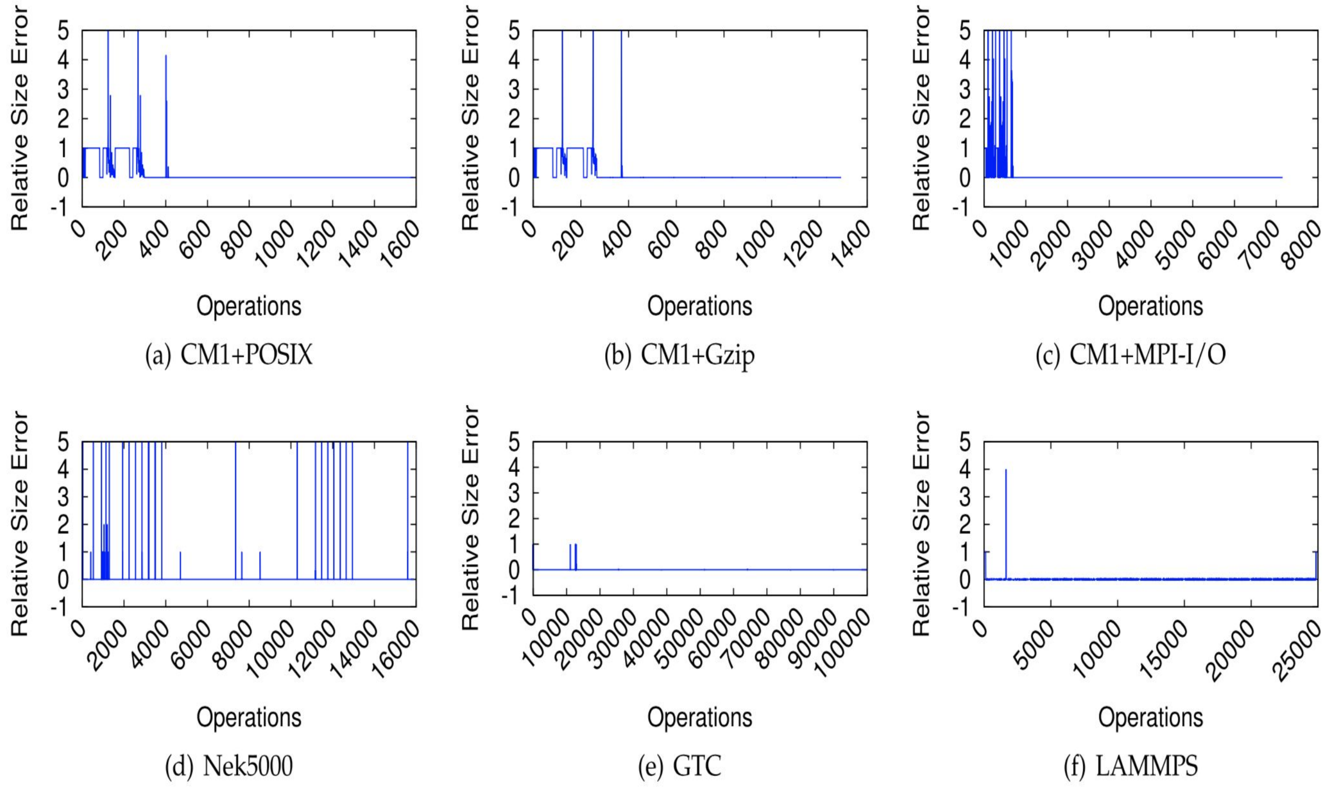


Fig. 5. Relative error in the prediction of access sizes in all simulations: $E_{size} = \frac{|size_p - size_o|}{size_o}$, where $size_p$ is the predicted size and $size_o$ is the actually observed size.

$$E_{size} = \frac{|size_p - size_o|}{size_o},$$

TABLE 6
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

Evaluation

Temporal Prediction -1

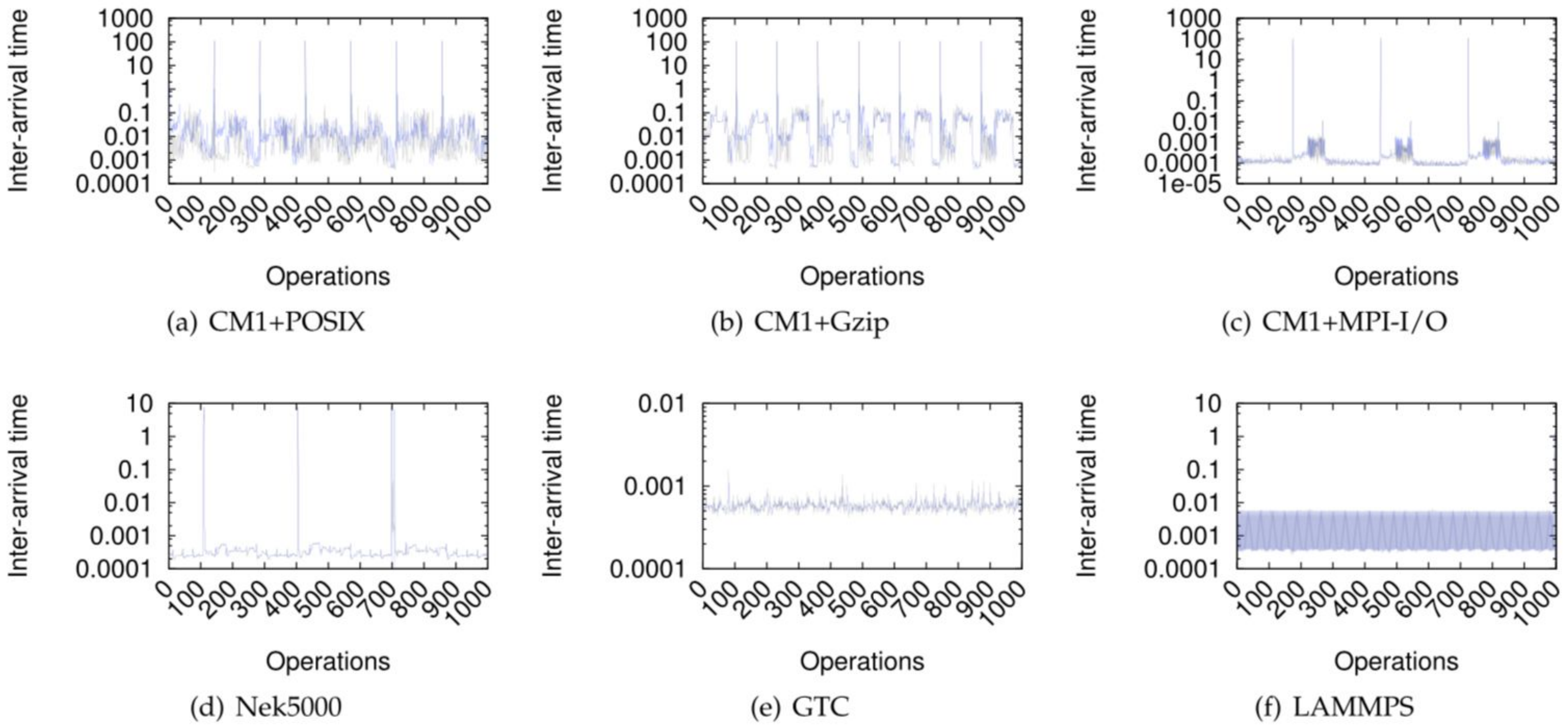


Fig. 7. Matching between observed (gray) and predicted (blue) interarrival times of I/O events.

Evaluation

Temporal Prediction -2

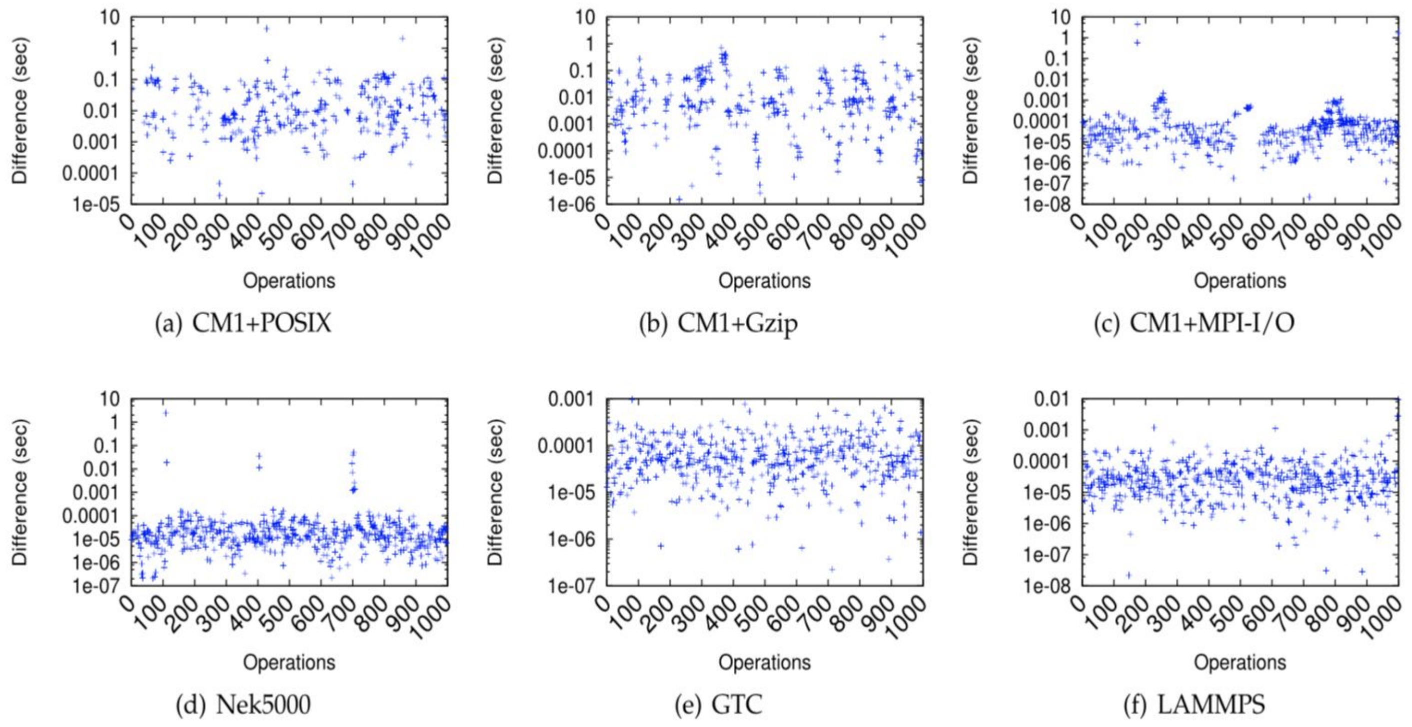


Fig. 8. Difference between predicted and observed interarrival times of I/O events.

TABLE 9
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	0.011 sec	0.049 sec
GTC	0.001 sec	0.006 sec
LAMMPS	0.000 sec	0.003 sec

Related Work

- Grammar-Based Modelling
  - Sequitur (repetitive periodic I/O)
- I/O Patterns Prediction
  - Spatial Predictions
    - Gniady (Improve caching)
    - Madhyastha and Reed (using Artificial Neural Networks
    - He (improve metadata indexing in PLFS, considers sequence of offset and size, uses LZ77 algorithm)
  - Temporal Prediction and Scheduling
    - Tran and Reed (using ARIMA time series)
    - Byna (can be used in later runs)
    - Zhang (trend toward smaller operating systems with only restricted features so not applicable in future machines with no preemptive process scheduler)

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 run-time 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 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. It is based on an extension of Nevill-Manning's Sequitur algorithm, that called StarSequitur, and is able to make accurate prediction of any N future I/O operations.

As future work, Omnisc'IO to integrate within CALCioM framework for efficient I/O scheduling and to implement prefetching and caching systems that leverage the excellent prediction capabilities shown by Omnisc'IO. There are also another 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.

Thank you for your time.

Any questions?