Improving Read Performance with Online Access Pattern Analysis and Prefetching

Houjun Tang[1,2], Xiaocheng Zou[1,2], John Jenkins[1,3], David A. Boyuka II[1,2], Stephen Ranshous[1,2], Dries Kimpe[3], Scott Klasky[2], and Nagiza F. Samatova[1,2,⋆]

[1] North Carolina State University, Raleigh, NC 27695, USA
[2] Oak Ridge National Laboratory, Oak Ridge, TN 37830, USA
[3] Argonne National Laboratory, Argonne, IL 60439, USA
samatova@csc.ncsu.edu

Abstract. Among the major challenges of transitioning to exascale in HPC is the ubiquitous I/O bottleneck. For analysis and visualization applications in particular, this bottleneck is exacerbated by the write-once-read-many property of most scientific datasets combined with typically complex access patterns. One promising way to alleviate this problem is to recognize the application's access patterns and utilize them to prefetch data, thereby overlapping computation and I/O. However, current research methods for analyzing access patterns are either offline-only and/or lack the support for complex access patterns, such as high-dimensional strided or composition-based unstructured access patterns. Therefore, we propose an online analyzer capable of detecting both simple and complex access patterns with low computational and memory overhead and high accuracy. By combining our pattern detection with prefetching, we consistently observe run-time reductions, up to 26%, across 18 configurations of PIO-Bench and 4 configurations of a micro-benchmark with both structured and unstructured access patterns.

1 Introduction

Scientists who work with simulations such as S3D combustion [1] and GTS core plasma fusion [2] spend a significant amount of time analyzing the massive amount of data generated. With the increasing gap between CPU and I/O, the performance of scientific analysis and visualization applications are often I/O-bound [3], thus read performance becomes a key area for optimization. An essential component of this process is to better understand the application's I/O behavior or its access patterns.

An access pattern is a sequence of accesses that exhibits a certain regularity. Many common access patterns occur as a result of iterative computations [4]. For example, if a matrix is stored in row-major format, reading consecutive rows of the matrix results in a contiguous pattern, whereas reading one column induces a simple-strided pattern with the file pointer incremented by the same amount (row size) between each request. Scientific applications exhibit these

⋆ Corresponding author.

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 246–257, 2014.

patterns and others, including higher dimensional strided access patterns and composition-based or correlation-based unstructured access patterns.

Recognizing access patterns in an application is a key to potentially reducing future file read time. Scientific applications often read and analyze data alternately, thus by overlapping the two phases with prefetching can significantly reduce the overall execution time of the application. Accurate prefetching can be achieved with access pattern analysis.

In order to achieve high prefetching accuracy, it is necessary to acquire comprehensive knowledge of the application's access patterns. Various methods have been proposed [5–8], however, these tools are all offline-based and not capable of detecting complex access patterns (such as composition-based unstructured access patterns). Offline-based tools assume access history of one or more previous runs beforehand, which is unrealistic to obtain for scientific applications nowadays that run for hours or even days. In addition, offline based algorithms cannot be directly applied to online analysis as 1) they assume the presence of full access history, which may not fit in the memory; and 2) they detect a pattern after its full occurrence, which provides no useful information for the current optimization strategy.

We propose a method for online analysis that requires no prior information of the application. To the best of our knowledge, our method is the first one capable of performing online analysis of various complex access patterns. The contributions of this work are as follows:

Online, Low-Overhead Pattern Analysis with High Accuracy. We adopt a "pattern growth" approach and efficient pattern detection algorithms to enable online analysis with overhead less than 5% in all test cases. The overall run-time reduction is up to 26% via pattern-aware prefetching with accuracy up to 99%;

Support for Various Access Patterns. We develop an analyzer capable of detecting structured access patterns as well as composition-based and correlation-based unstructured access patterns;

Low Memory Footprint. To retain low memory footprint during run-time, we merge I/O traces with their corresponding access patterns in a compact format and keep a limited number of recent trace records in memory.

2 Background

Many I/O access patterns classification approach have been proposed [5, 9, 10]. Compared with them, we additionally support unstructured access pattern. Although the access pattern classification is similar, the algorithms to detect the patterns are different for offline and online analysis, as explained in Section 3.1.

2.1 Structured Access Pattern

Structured access patterns include contiguous, simple-strided, and kd-strided patterns. Fig. 1 illustrates the former two kinds. A contiguous pattern occurs when consecutive read requests are accessing a contiguous region of data in

a file. It can be further divided into uniform and variable size patterns. For strided patterns, a stride is the difference between starting offsets of consecutive requests, and is fixed within each dimension. Simple-strided pattern is a special case of kd-strided when $k = 1$. A kd-strided pattern can be viewed as a series of $k - 1$d-strided segments with its k dimensional stride. For example, a 2d-strided pattern with the following offsets: {1, 3, 5, 11, 13, 15, 21, 23, 25}, is composed of three simple-strided segments {1,3,5}, {11,13,15}, and {21,23,25}, with the second dimensional stride of 10. Kd-strided pattern is often found when accessing a sub-volume or sub-plane of multi-dimensional data.

2.2 Unstructured Access Pattern

Unstructured access patterns are accesses that exhibit patterns with less regularity compared to structured ones. The number of accesses is linear to the number of parameters representing them, while exponential for structured ones. There are two particular instances that we found useful for scientific applications, which are referred as composition-based and correlation-based unstructured access patterns. The composition-based patterns capture the repeating intervals between structured patterns or individual accesses, which is further explained in Section 3.3. Previous research in [7] exploited block correlations in storage systems. We include this kind of pattern and referred it as correlation-based unstructured access patterns. For example, from an offset sequence of {10, 20, 30, 40, 50, 10, 70, 20, 30, 80, 10, 40, 20, 30}, the correlation-based pattern is $\{10|\,20, 30\}$ and $\{20|\,30\}$, meaning that the data starting from offset $20, 30$ is frequently accessed after 10, while 30 is often accessed after 20. The threshold value of frequent accesses is 3, which is the number of times an offset occurs to be considered in a pattern. The request size is omitted for simplicity.

3 Method

Our online analyzer performs access pattern analysis of applications during their run-time and utilize the pattern information to guide prefetching for better performance. Fig. 2 illustrates the overview of our framework.

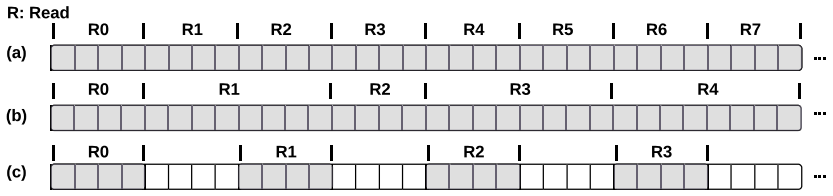

Fig. 1. Each block represents 1 byte of data stored in row-major format, with shaded blocks being accessed. (a) Contiguous with uniform size: 8 requests ($R0$ to $R7$) each access 4 bytes. (b) Contiguous with variable size: 5 requests with sizes of $4, 8, 4, 8, 8$ bytes. (c) Simple-strided: 4 requests each access 4 bytes of data with 8 bytes between the starting offsets of consecutive requests.

3.1 Online Access Pattern Analysis

We adopt a rule-based model for access pattern detection in our online analyzer, which is the key component the framework. We maintained a "pattern library" that contains a collection of rules. These rules provide a concise description of the access sequences that are recognized as access patterns. The input is a sequence of accesses and the output is the detected access patterns and corresponding prefetching instructions.

Each time a read request is traced, the analyzer first performs a lookup in the pattern history to decide whether to activate a previously detected pattern and start prefetching or use it for analysis. The pattern analysis procedure includes the following steps: 1) create a new pattern if current records in the trace buffer match any detection rules in the pattern library; 2) "grow" the current pattern if the following accesses belong to it and inform the prefetcher to prefetch data that are predicted to be accessed next; 3) commit the access pattern to the pattern history when the new access do not fit in; 4) attempt to coalesce the current pattern with previous structured ones to form a higher level pattern; 5) look back in the pattern history and check if there is any pattern that matches the current one. More details of this procedure are explained in later examples of structured and unstructured pattern analysis.

Unlike offline analysis with a complete access history, online analysis must be incremental to detect a pattern during its occurrence. Thus we adopt the above "pattern growth" approach: as new accesses arrive, they are compared to the current active pattern before being inserted to the trace buffer. The pattern library consists of detection and coalesce rules for detecting structured and unstructured access patterns. The difference between them are the objects they operate on: detection rule operates on offset of accesses while coalesce rule operates on patterns. The analysis is performed periodically instead of upon every new request to reduce computation overhead. Three threshold values (T_{struct}, T_{corr}, and T_{comp}) are used to trigger the analysis of structured, correlation-based unstructured, and composition-based unstructured access patterns.

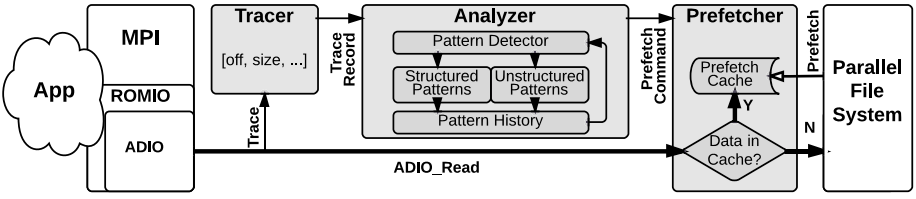

Fig. 2. An overview of our framework: each time a read request is made, the tracer extracts the read request's information while it is being passed to the prefetcher. The requested data are copied to user buffer if found in the prefetch cache or a normal file read is issued to the parallel file system, the components added are in shaded shapes.

3.2 Structured Access Pattern Analysis

Different detection rules are used for contiguous and simple-strided access patterns. A contiguous pattern is determined by having at least 3 consecutive accesses with no gap in between. A simple-strided pattern comes with same offset differences (stride) between at least 3 consecutive accesses with identical request size. Kd-strided pattern is composed of $(k-1)$d-strided segments and is detected by the coalesce rule, which checks the stride and the number of accesses of two strided patterns with the same dimension. Note that each dimension of a kd-strided pattern must have at least three $(k-1)$d-strided segments.

Take a 2d-strided pattern with the following offsets {1, 3, 5, 11, 13, 15, 21, 23, 25, 31, 33, 35} as an example. The second dimensional stride can not be determined until 31 is accessed that signals the end of the third simple-strided segment. With three simple-strided segments detected and committed, they are coalesced to a 2d-strided one(step 1 to 4 of the pattern analysis procedure). An earlier detection is possible if a previous 2d-strided pattern with the same stride and number of accesses of first dimension exists in the pattern history: we temporarily mark the current simple-strided pattern of $\{1, 3, 5\}$ as the 2d-strided one and start prefetching (step 5). Once a mismatch happens, it is restored to the previously detected pattern and continue the analysis procedure. Only the most recent pattern that qualifies is used in case multiple candidates exist, as same pattern tends to occur close in time. The time complexity for detection rule is $O(n \times T_{struct})$, and for the coalesce rule is $O(N_{spattern})$, where n is the number of total accesses, and $N_{spattern}$ is the number of detected structured access patterns. Though the time complexity depends on the whole trace and could be quite large, the frequency of the analysis is expected to be high and as a result for each analysis procedure the workload is relatively small.

3.3 Unstructured Access Pattern Analysis

Previous analyzers usually deal with access patterns build from individual accesses. However, when accessing time-series data generated by scientific simulations, a higher level of pattern often exists between the accesses of different time steps. For example, if a scientist wants to visualize a climate dataset with hourly recorded data at the times when the daily low/high temperature occurs (usually 5-6am and 2-3pm) for 30 days. The corresponding visualization application would read data of time step 5, 6, 14, 15, 29, 30, 38, 39, 53, 54, 62, 63, etc. and for each time step, structured access patterns could exist if a sub-volume decomposition is used for parallel processing. State-of-the-art analyzer like IOSIG [5] is only able to detect the structures ones within each time step, while not recognizing the higher level of composition-based unstructured pattern with time step intervals repeating 29 times of $\{1, 8, 1, 14\}$.

The detection rule for composition-based pattern detection is to find offset delta (the difference between any two consecutive offsets) sequences that repeat at least twice. Two separate delta sequences are created from the offset of accesses and the starting offset of structured access patterns. To efficiently detect

such patterns, we build suffix trees incrementally that has linear time and space complexity. The corresponding pattern can be easily obtained from its suffix tree after each time of analysis.

For correlation-based access patterns, steps 2 and 4 are skipped because a correlation-based pattern stays the same once generated. In step 5, patterns are merged into one if a previous pattern with the same "entry" is found. Only accesses with request size larger than R_{size} are considered because the cost of analyzing those accesses outweighs the cost brought on by prefetching. In addition, we only focus on frequent accesses (occurs more than T_{freq} times) with their next N_{next} accesses. And the time complexity is $O(n \times N_{next})$ The frequent access is referred as the "entry" of a pattern. A candidate set of accesses that have the potential of becoming frequent, which have a frequency between $T_{freq} - \epsilon$ and T_{freq}, is maintained for incremental analysis. The analyzer then forms the pattern of each frequent access as the entry and a list of its following frequent accesses. Each time the entry is accessed, this pattern is activated and the following accesses are prefetched as much as possible.

3.4 Trace Storage with Low Memory Footprint

Our framework requires limited additional memory usage during application's run-time. The tracer extracts useful information from read requests and passes them to the analyzer to determine whether to store them in the trace buffer. Trace records are compressed to a pattern representation if possible. The memory used for structured access patterns are significantly reduced due to its regularity. A 2d-strided pattern with 1024 accesses needs approximately 102KB in memory while only 134B with a pattern representation. The unstructured access patterns require more storage than structured but still use much less memory than keeping all its accesses. In addition, since online analysis focuses on current access patterns, only recent trace records are kept in the trace buffer. The tracer is implemented in the ADIO layer of MPI-IO, on which MPI optimizations like data sieving can be captured and utilized, as well as allowing the usage of other PMPI-based methods, such as Darshan [11].

3.5 Informed Prefetching

The prefetcher prefetches data informed by the analyzer and checks if data in the prefetch cache can be used for current request. Depending on the accecc pattern, the size of prefetched data varied, and we only consider relatively large data size ($> 1KB$) as smaller request sizes do not benefit from prefetching. It is also implemented in ADIO layer and prefetches data per MPI process using a prefetching thread. To avoid extra overhead caused by communication between processes, both the analysis and prefetching are per-process based. We adopted a conservative prefetching strategy to minimize the cost of mis-prefetching: the prefetcher starts to prefetch data when a stable access pattern is detected and stops immediately when the previously prefetched data is not used, which indicates the detected access pattern is terminated.

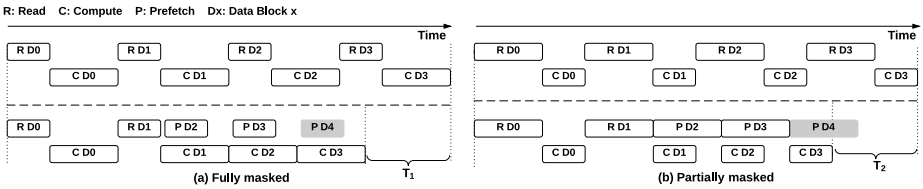

Fig. 3. Prefetching is fully and partially masked by computation

4 Experimental Results

4.1 Experimental Setup

Our experiments were conducted on Argonne LCRC Fusion cluster. Each node is equipped with Intel Xeon 8-core (dual quad-core) 2.53 GHz processor, 36 GB memory, and 250GB local disk. The attached local disk to each node enables us to set up our own PVFS2 servers and create an isolated environment. We used 8 server nodes running PVFS2 2.8.2 file system with default strip size of 64KB. These nodes are connected with InfiniBand QDR and Gigabit Ethernet. Additionally, we implemented our framework based on MPICH 3.0.4.

4.2 Structured Access Pattern Performance

We used the PIO-Bench [12], a widely used synthetic parallel file system benchmark suite, and conducted experiments with contiguous, simple-strided, and 2d-strided access patterns to evaluate the performance with structured access pattern detection.

As mentioned in Section 1, the benefit of prefetching comes from overlapping I/O and computation. Fig. 3 illustrates four periodic read (R D0 to R D3) that are fully and partially masked by the computations via informed prefetching and the total time of T_1 and T_2 is reduced. To mimic real application's behavior, we insert computation time between each file read operation of PIO-Bench. To determine the computation time, we collected the time of running GNU Scientific Library functions such as find minimum number, first 100 smallest numbers, mean, standard deviation, and sorting. The ratio of computation time to read time for different size of data are shown in Table 1. We found the ratio of 0.5, 1.0, and 2.0 could represent different scenarios of real computation time and thus are used in our experiments. The results of simple-strided is similar to those of 2d-strided and due to space limitation, we only show the results using ratio of 0.5 and 1.0 that represent I/O intensive and compute intensive scenarios, contiguous and 2d-strided access pattern, and read request of 128KB and 1MB.

From the results shown in Fig. 4 we can see a reduction in the application's total running time in all cases with the percentage of up to 26% and an average of 17% for contiguous access pattern and 16% for 2d-strided. The performance gain of the informed prefetching with access pattern analysis are more pronounced

Table 1. Ratio of computation time to read time for a given size of data

Size	min	min100	mean	sd	sort
128KB	0.027	0.061	0.183	0.353	1.388
1MB	0.028	0.031	0.221	0.428	1.899
16MB	0.034	0.027	0.244	0.473	2.586

Table 2. Prefetching accuracy of three structured access patterns

Pattern Type	Size	Read #	Accuracy
Contiguous / Simple-strided	128KB	1024	99.9%
	1MB	512	99.8%
	16MB	32	96.5%
2d-strided	128KB	1024	99.8%
	1MB	512	99.6%
	16MB	32	92.0%

when the computation to read time ratio is 1.0 because read time is fully masked by computation. For ratio with 2.0, the time reduction percentage is between that of 1.0 and 0.5, which is expected because the potential of run-time reduction is less when computation takes most of the time.

4.3 Unstructured Access Pattern Performance

The random strided pattern of PIO-Bench is a composition-based unstructured access pattern, however, this pattern is too simple compared to real scientific applications. Thus we developed a micro-benchmark with both structured and unstructured access patterns. We found the results for correlation-based patterns are similar to those in [7] and thus it is not included in our micro-benchmark. The micro-benchmark simulates the file read behavior of an application mentioned in Section 3.3, which performs 3D visualization of climate datasets with hourly data at time steps when daily low/high temperature occurs. A sub-volume decomposition is used to perform parallel I/O for each time step. We experimented with two types of decompositions: row-wise and column-wise, as shown in Fig. 5. For each time step, the 3D data is broke into "slices" and each process reads one slice. The resulting access pattern contains both structured (simple/2d-strided within each time step) and composition-based unstructured pattern (repeating kd-strided with time step interval rotates from $\{1, 8, 1, 14\}$). Similar to the previous experiments, we set the computation time to the average time of each file read. In addition to using plain row-major file layout, we also tested with files stored with block layout. Scientific applications like ScaLAPACK benefits from this kind of layout as they use blocks as the unit for communication and computation. The normal row-major file layout can also be viewed as the block layout with block size of 8B (the size of double).

The total data size of each time step read by all processes is 1GB and we vary the decomposition type, file layout type, and the number of processes. All processes are synchronized before the first read and the maximum elapsed time is reported. Fig. 6 compares the performance results by row and column decomposition with different file layout types. The row decomposition of different block sizes have similar results, and column decomposition with row-major layout takes much longer time since it has most dis-contiguous accesses, and are omitted due to space limitation. For all cases, we observe the time reduction ranges from 13% to 26% with prefetching, which proves the effectiveness of the analyzer.

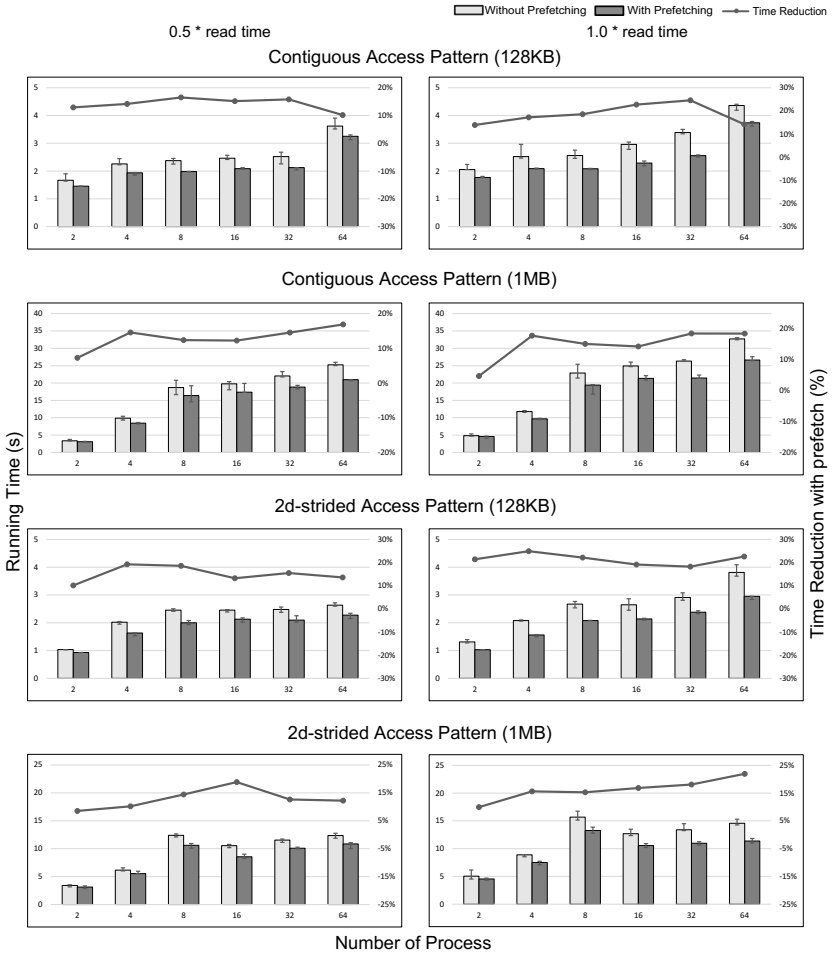

Fig. 4. Performance of contiguous and 2d-strided access patterns

4.4 Overhead of Trace Collection and Access Pattern Analysis

The overhead of our trace collector and analyzer is defined as the time difference between the two runs with our framework and with original MPICH. To test the overhead of trace collection and analysis, we run with the previous configurations by setting the computation time to, the median of 10 different runs is used. Due to space constraint, we only show results of two different cases in Fig. 7. Similar overhead is observed in other cases and all are less than 5%.

4.5 Accuracy of Access Pattern Detection

To evaluate the effectiveness of our pattern detection algorithm, we use prefetching accuracy as a metric. It is calculated by dividing the amount of subsequently

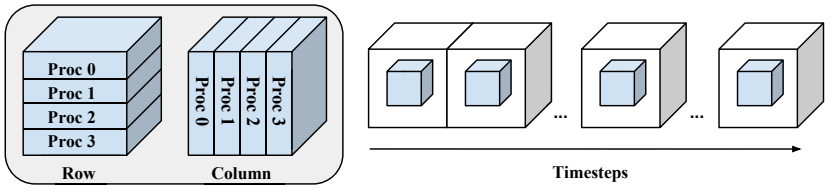

Fig. 5. Two types of domain decomposition used in our evaluation

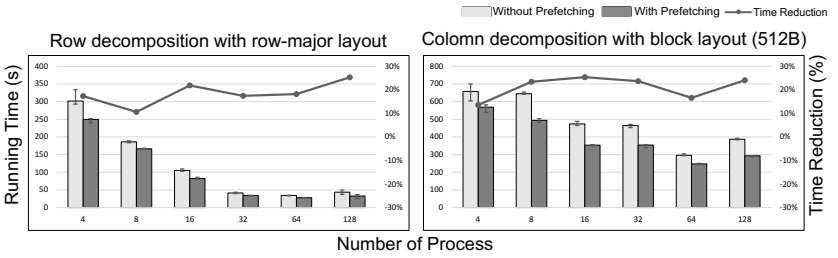

Fig. 6. Performance of row/column domain decomposition with different block size

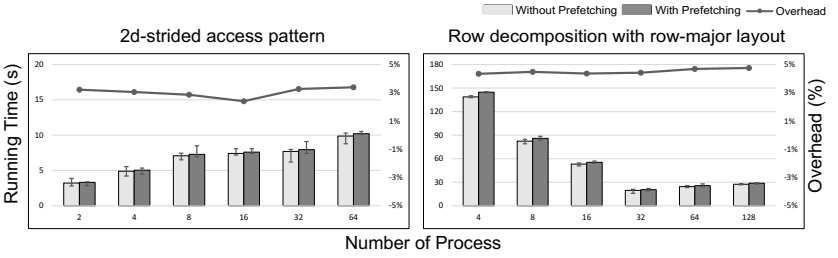

Fig. 7. Overhead of trace collector and analyzer with 2d-strided and unstructured access pattern

used and prefetched data by the total used data. The high accuracy means the prediction of analyzer is accurate. Table 2 shows the prefetching accuracy of three structured access patterns. The high percentage is expected as these patterns are highly structured and remain stable for a period of time.

5 Related Work

Various methods have been proposed to utilize access patterns for I/O optimization. Gong et. al [13] proposed a parallel run-time layout optimization framework to speed up queries on large complex scientific datasets. In database community, utilizing access patterns to guide prefetching proves to be effective [14]. Unlike their methods that deal with file layout organization and database objects, respectively, our work only involves MPI-IO and is on byte level.

Most of the existing pattern analyzers [5–8] perform analysis in the offline-based fashion. In [5], a notation called I/O signature that represents access

patterns is proposed. However, their pattern analysis only focus on structured ones. Oly et al. used a Markov model [6] built from access history to predict future accesses and prefetch data. C-Miner [7] uses a frequent sequence mining algorithm named CloSpan to discover block correlations, and utilizes the detected information for prefetching and reorganizing data layout. Choi et. al applied probabilistic latent semantic analysis with deterministic annealing [8] to discover file or variable access patterns. These methods require prior knowledge of the application and can not be directly applied to online analysis. We enabled our trace collection and analysis to be online, which is more desirable for scientific applications nowadays. Our analyzer can also be used in offline manner that generates same access patterns as offline-based ones.

Prefetching is an effective latency-hiding solution for improving efficiency of parallel I/O and has been extensively studied and widely used [15–18]. However, the traditional prefetching strategies such as file-system level approaches are conservative. Even with advanced parallel file systems such as PVFS [19] and Lustre [20], high bandwidth is not achieved when only simple patterns such as contiguous or simple strided are detected. They cannot provide satisfactory performance for the modern scientific simulations with a large number of complex access patterns. Patterson et al. proposed informed prefetching [21], but this requires developers to add I/O hints to the program. Unlike their method, our framework requires no code modification.

6 Conclusion

We proposed an online access pattern analyzer that supports both structured and unstructured access patterns with high accuracy and low computation and memory overhead. With the pattern-aware prefetching, our method results in up to 26% run-time reductions on top of less than 5% overhead with both kind of access patterns in 22 benchmark evaluations.

Acknowledgements. We would like to thank the Leadership Computing Facilities at Argonne National Laboratory and Oak Ridge National Laboratory for the use of resources. Oak Ridge National Laboratory is managed by UT-Battelle for the LLC U.S. D.O.E. under Contract DE-AC05-00OR22725. This work was supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and the U.S. National Science Foundation (Expeditions in Computing and EAGER programs).

References

1. Chen, J.H., Choudhary, A., De Supinski, B., DeVries, M., Hawkes, E., Klasky, S., Liao, W., Ma, K., Mellor-Crummey, J., Podhorszki, N., et al.: Terascale direct numerical simulations of turbulent combustion using S3D. Computational Science & Discovery 2(1), 15001 (2009)

2. Wang, W., Lin, Z., Tang, W., Lee, W., Ethier, S., Lewandowski, J., Rewoldt, G., Hahm, T., Manickam, J.: Gyro-kinetic simulation of global turbulent transport properties in tokamak experiments. Physics of Plasmas 13, 092505 (2006)
3. Zhu, Y., Jiang, H., Qin, X., Feng, D., Swanson, D.R.: Improved read performance in a cost-effective, fault-tolerant parallel virtual file system (ceft-pvfs). In: CCGrid 2003, pp. 730–735. IEEE (2003)
4. Di Biagio, A., Speziale, E., Agosta, G.: Exploiting thread-data affinity in openmp with data access patterns. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 230–241. Springer, Heidelberg (2011)
5. Byna, S., Chen, Y., Sun, X.H., Thakur, R., Gropp, W.: Parallel I/O prefetching using MPI file caching and I/O signatures. In: SC 2008, pp. 1–12. IEEE (2008)
6. Oly, J., Reed, D.A.: Markov model prediction of I/O requests for scientific applications. In: ICS 2002, pp. 147–155. ACM (2002)
7. Li, Z., Chen, Z., Srinivasan, S.M., Zhou, Y.: C-Miner: Mining Block Correlations in Storage Systems. In: FAST, pp. 173–186 (2004)
8. Choi, J.Y., Abbasi, H., Pugmire, D., Podhorszki, N., Klasky, S., Capdevila, C., Parashar, M., Wolf, M., Qiu, J., Fox, G.: Mining hidden mixture context with adios-p to improve predictive pre-fetcher accuracy. In: 2012 IEEE 8th International Conference on E-Science (e-Science), pp. 1–8. IEEE (2012)
9. Crandall, P.E., Aydt, R.A., Chien, A.A., Reed, D.A.: Input/output characteristics of scalable parallel applications. In: Proceedings of the IEEE/ACM SC 1995 Conference on Supercomputing, pp. 59–59. IEEE (1995)
10. Madhyastha, T.M., Reed, D.A.: Learning to classify parallel input/output access patterns. TPDS 13(8), 802–813 (2002)
11. Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., Riley, K.: 24/7 characterization of petascale I/O workloads. In: Cluster 2010, pp. 1–10 (2010)
12. Shorter, F.: Design and analysis of a performance evaluation standard for parallel file systems. PhD thesis, Clemson University (2003)
13. Gong, Z., Boyuka, D., Zou, X., Liu, Q., Podhorszki, N., Klasky, S., Ma, X., Samatova, N.F.: Parlo: Parallel run-time layout optimization for scientific data explorations with heterogeneous access patterns. In: CCGrid 2013, pp. 343–351 (2013)
14. Han, W.S., Moon, Y.S., Whang, K.Y.: Prefetchguide: Capturing navigational access patterns for prefetching in client/server object-oriented/object-relational dbmss. Information Sciences 152, 47–61 (2003)
15. Baer, J.L., Chen, T.F.: An effective on-chip preloading scheme to reduce data access penalty. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing 1991, pp. 176–186. IEEE (1991)
16. Dahlgren, F., Dubois, M., Stenstrom, P.: Fixed and adaptive sequential prefetching in shared memory multiprocessors. In: ICPP 1993, vol. 1, pp. 56–63. IEEE (1993)
17. Dahlgren, F., Dubois, M., Stenstrom, P.: Sequential hardware prefetching in shared-memory multiprocessors. TPDS 6(7), 733–746 (1995)
18. Ding, X., Jiang, S., Chen, F., Davis, K., Zhang, X.: Diskseen: Exploiting disk layout and access history to enhance I/O prefetch. In: USENIX Annual Technical Conference, vol. 7, pp. 261–274 (2007)
19. Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, R.: Pvfs: A parallel file system for linux clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference, pp. 391–430 (2000)
20. Braam, P.J., Zahir, R.: Lustre: A scalable, high performance file system. Cluster File Systems, Inc. (2002)
21. Patterson, R.H., Gibson, G.A., Ginting, E., Stodolsky, D., Zelenka, J.: Informed prefetching and caching, vol. 29. ACM (1995)