KNOWAC: I/O Prefetch via Accumulated Knowledge

Jun He, Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
Chicago, Illinois 60616
{jhe24, sun}@iit.edu

Rajeev Thakur
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439
thakur@mcs.anl.gov

***Abstract*—The lasting memory-wall problem combined with the newly emerged big-data problem makes data access delay the first citizen of performance optimizations of cluster computing. Reduction of data access delay, however, is application dependent. It depends on the data access behaviors of the underlying applications. Therefore, leaning and understanding data access behaviors is a must for effective data access optimizations. Modern microprocessors are equipped with hardware data prefetchers, which predict data access patterns and prefetch data for CPU. However, memory systems in design do not have the capability to understand data access behaviors for performance optimizations. In this study, we propose a novel approach, named KNOWAC, to collect I/O information automatically through high-level I/O libraries. KNOWAC accumulates I/O knowledge and reveals data usage patterns by exploring the collected high-level I/O characteristics. The discovered data usage patterns can be used for different I/O optimizations. We apply KNOWAC to I/O prefetch under the framework of PnetCDF in this study. Experimental results on a real-world application show that KNOWAC is promising and has a true practical value in mitigating the I/O bottleneck.**

I. Introduction

Big data provide unprecedented opportunities for scientific discoveries. With timely access and analysis of data produced by simulations and instruments, scientists can gain new insights faster and accelerate their research and discovery. In the meantime, however, big data pose a great challenge on I/O systems as well [1] [2] [3]. The computing power has been growing following the Moore's Law during the past decades, but I/O performance does not keep up the pace. Big data requirements and limited I/O capability have made I/O the bottleneck of today's high performance computing systems. To address this problem, parallel I/O techniques have been developed to improve I/O performance. MPI-IO library [4] provides uniform I/O interfaces for parallel data accesses. Parallel file systems (such as PVFS [5], Lustre [6] and GPFS [7]) have been deployed to enhance the data access parallelism. These techniques provide a general solution, but how to utilize the underlying parallelism is still application and environment dependent. Current I/O stacks are stateless. They cannot collect, store or use application-specific characteristics which are essential for I/O performance optimizations. While I/O becomes a determine factor of system performance, collecting and using past information become a natural choice of understanding I/O behavior for future performance.

In order to take advantage of application-specific characteristics, we propose a stateful I/O stack with the ability to accumulate knowledge of applications and machines. This approach is named KNOWAC (KNOWledge ACcumulation). KNOWAC I/O system gathers and uses application-specific characteristics to improve I/O performance in an application-aware manner. The idea behind KNOWAC is that the computation model and data usage model of an application is relatively stable. That means the high-level I/O patterns of each application is relatively steady. This is true for most real-world applications. As long as we can carefully handle branching, the computation and workflow structure of an application is quite steady and does not change with input data. By collecting and analyzing high-level I/O behaviors and associated control flow information, application-specific I/O patterns can be identified and used for optimizations.

One of the potential optimizations is data prefetching, which fetches data ahead of time to hide the gap between I/O and computing capabilities. Timing and accuracy are two important factors which determine the effectiveness of prefetching. Prefetching wrong data or fetching right data but at a wrong time does not help and often hurts the performance. To prefetch effectively, the proposed stateful I/O system predicts future accesses by following the high-level I/O patterns in KNOWAC knowledge repository of the application.

We designed and implemented the KNOWAC prefetching system under the framework of PnetCDF [8]. PnetCDF is a popular high-level I/O library in which data variables are accessed by logical names. The logical names offer us the opportunity to collect and analyze high-level I/O patterns under PnetCDF. The high-level I/O patterns are stored as graphs in a knowledge repository. The graphs describe the time and dependency relationships among data variables. KNOWAC prefetching system assists prefetching by matching the run-time I/O behaviors with the pre-stored patterns. If a match is found, the system conducts prefetching by following the matched pattern. The knowledge graphs are adjusted and refined with new information whenever the application runs. In this way, KNOWAC provides a better optimization for frequently used applications.

Our work has the following contributions.

1) A new mechanism is introduced to analyze I/O behaviors and identify I/O patterns via high-level I/O libraries.
2) A stateful I/O stack is proposed to make I/O system smarter. Graph representations are used to accumulate application-specific knowledge and to make intelligent decisions based on past I/O behaviors.
3) An effective prefetching system is developed based on the stateful I/O stack. This system is transparent to the user and simple in implementation, which makes it practically valuable.

The rest of this paper is organized as follows. In Section II, we briefly review related works. The overview of the KNOWAC system is introduced in Section III. In Section IV, we present the knowledge accumulation process. Analysis of application behaviors and the idea of knowledge accumulation are discussed. Section V provides implementation details. Section VI uses a real-world scientific application to verify the KNOWAC approach. Performance results (with different inputs, operations, storage devices) and overhead are presented and analyzed. Section VII concludes this study and discusses future work.

II. Related Work

Prefetch is a well studied technique. It improves data access performance by bringing data closer to computing. A computer system can conduct profetching at many levels, including cache, memory, and I/O. We focus on I/O prefetching in this study. Speculative execution [9] is a prefetching technique that is used to help prefetching for disk accesses. It takes advantage of the stall cycles of a process and speculatively executes the instructions in order to run beyond the original process and fetch the data ahead. Patterson et al. proposed informed prefetching [10] [11] [12], which puts the burden on the developers to add I/O hints to the program. The hints are passed down to the file system to assist prefetching. Informed prefetching and speculative execution merged together later in their research. Speculative execution was used to automatically generate hints for informed prefetching on disk [13]. Recently, Chen et al. proposed an approach to prefetch on modern I/O libraries and parallel file systems [14]. It uses pre-execution to provide necessary hints for prefetching to hide I/O latency.

Access history is also used to assist prefetching. Byna et al. proposed a notation named I/O signature to describe the historic access patterns in the offset-and-length level [15]. Prefetching is conducted based on the information in the notation in the later runs. Oly et al. uses Markov model [16], which is built with access history, to predict future accesses and prefetch data for scientific applications. It exploits spatial access patterns at a low level. An analytical approach was proposed by Lei et al. to analyze interesting system events at run time, find out correlations of file accesses and then use it to prefetch data [17]. DiskSeen [18] exploits temporal and spatial relationships of disk blocks by access histories and uses that information to prefetch.

Cache is an important component of a prefetching system, since prefetched data is stored in it. Many articles focus on caching. A scheduling algorithm for prefetching at disk level is discussed in [19]. Kallahalla et al. proposes I/O delegated nodes to improve the scalability of a caching system [20]. Cache was used on I/O delegated nodes to reduce the number of requests to the disk. dCache [21] is a cache system that implements a file cache for all compute nodes. All the nodes access the cache using a uniform name space as in a traditional shared memory machine. Kallahalla et al. studies the cache management and prefetch scheduling in [19].

The techniques listed above conduct prefetching without the semantics of the data. Therefore they cannot take advantage of the high-level usage patterns to predict future accesses and prefetch aggressively. The approach presented in this paper exploits the logical semantic locality instead of the traditional spatial and temporal locality in the offset-and-length level. The high-level knowledge captured makes application-aware optimizations possible.

III. System Overview

KNOWAC prefetching strategy is designed for typical high performance cluster systems as shown in Figure 1. In these systems, compute nodes and I/O nodes are connected by a network. Applications in the compute nodes access data by calling high-level I/O libraries, such as Parallel NetCDF and Parallel HDF5. The high-level I/O libraries actually use MPI-IO to perform parallel I/O. Our design is general enough to be applied to systems with different architectures. For example, systems like BlueGene have dedicated I/O nodes upon which KNOWAC prefetching can be implemented to shorten the I/O latency [22] [23].

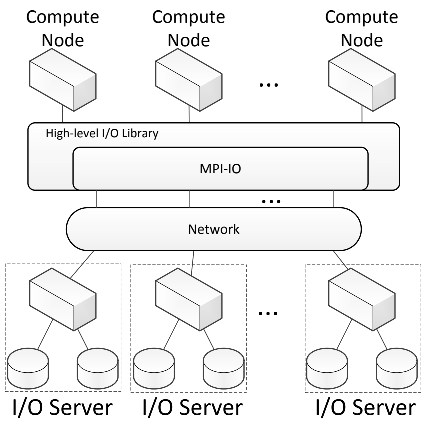


Fig. 1. Overview of a system using high-level I/O library.

Figure 2 shows the software architecture on one compute node. Whenever the applications access data via the high-level I/O library, the behaviors are collected and knowledge of the accesses is accumulated, analyzed and stored in a knowledge repository. The knowledge includes logical names of the data variables, the I/O operations, the time of accessing, etc. When the knowledge repository has enough information to reflect the characteristics of the application, the prefetch helper thread starts to prefetch data to cache.The application's main thread then can read data from cache directly if the data have been prefetched on time. The key challenge of the proposed prefetching system is the knowledge accumulation component which gathers application-and-machine-specific knowledge for prefetching. High-level global knowledge of I/O accesses is the core of KNOWAC.

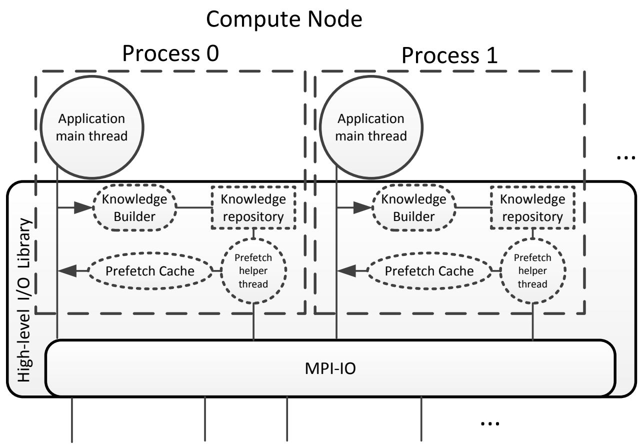


Fig. 2. Software architecture in a single node.

	R		W	
R	R R	R *R	R W	R *W
	*R R	*R *R	*R W	*R *W
W	WR	W *R	W W	W *W
	*W R	*W *R	*W W	*W *W

Fig. 3. Possible I/O behaviors

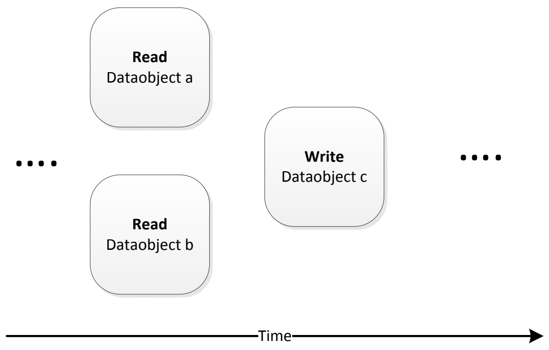


Fig. 4. An "R R" example

IV. Knowledge Accumulation

Most applications have their unique computation models or data processing procedures. These models and procedures do not change much from run to run. For example, MILC is a Lattice QCD(Quantum ChromoDynamics) program in which each sub-domain always exchanges data with its eight neighbors in a 4D space and then conducts computation independently [24]. MapReduce [25] applications often read data with the same format and then follow the same procedure of data analysis or visualization. Scientific applications with these features lie in the fields of cosmic science [26], earth science [27], climate science [28], simulation result analysis, visualization and so on. These relatively fixed computation models or data processing procedures imply relatively fixed I/O patterns. That is: at certain time of the computation, before or after certain events, applications need input or output data to conduct the next step. This characteristic can be used to predict the future I/O accesses and thus can be used to prefetch effectively.

A. High-level I/O Patterns

KNOWAC system records and analyzes I/O behaviors to find the unique logical characteristics of an application. Low level I/O libraries cannot achieve this goal. Because at a low level, data, in terms of offsets and lengths, are represented in bytes, which carry little logical meaning of the data. Moreover, the logical meaning of data at an offset may not be static. The file system abstraction does not require a data offset to be associated with any meaning. On the other hand, the high-level I/O libraries naturally have semantic information, which may imply how the data will be used with. For example, humidity and temperature may be used together to explore these relationship of them in a city. Then the data about the relationship may be used with wind velocity for further explorations. While the underlying offset-length behaviors might be different when accessing these data, the logical relationships among these three variables in the computation are always the same. KNOWAC takes advantage of the relationships to optimize I/O system performance. Data relationship involves time sequences of accessing data, combinations of data accessed, and relations between data and computation phases. I/O behaviors are studied in the following to determine I/O patterns.

Figure 3 shows all sets of two possible consecutive I/O behaviors. 'R R' represents a repeating pattern that the application reads the same two data variables each time it runs.

$$c = a + b;$$
$$c = c * b;$$

Assume a, b and c are big-size data objects. The timeline of the I/O operations for the codes above can be described by Figure 4. The information we get from I/O behaviors is $f(a, b) = c$, without considering the computation details. It is a computation model, or a data dependency relation. The goal of knowledge accumulation is to find such computation models and improve over time.

'R *R' means that the application reads the same data and then reads different data in different runs. For example, the application first needs to read an array to find out which part of another big array it should read next. In this case, it needs to read the same data and then read different parts of another data. This is a common case. For example, in HDEOS [27], it reads an array to find out the longitude and latitude boundaries of the area it needs. Then it reads that part of data from another array. The 'R *R' pattern could imply similar computation procedures. '*R R' may indicate the input of a procedure to get data for a function that needs multiple data objects. It is the same as 'R R' with some variety in the first read. These all-read cases are most likely used to get input for computation without outputting data to external storage.

The cases with a read as the first operation and a write as the second (e.g. 'R W', 'R *W') are likely to be the step of a computation phase where the application reads data, computes for a while and writes the results to an external storage. The cases with a write as the first operation and a read as the second (e.g. 'W R', '*W R') are likely to be the first step of a computation phase where the output from the last phase is ending and new data is needed to conduct the next phase. The cases with only writes (e.g. 'W W') are likely to be inside the procedure in which results are being written to different places. In an application, the purposes of writing could be to record new results or update existing results.

There are several observations that help us identify the data dependencies by looking at the high-level I/O behaviors. The first one is that when time interval of several reads are very close, they are likely to be the input of the same computation phase. This is reasonable because the principle of programming is read when it needs. No one would fill up memory with data that will not be used for a long time. Another observation it that the results of a computation phase are written out right after the computation phase.

B. The accumulation graph

We propose a knowledge graph to describe high-level I/O patterns and the relationships between data. In the graph, we use vertices to represent data objects, each of which contains data of the same logical meaning. For example, a data object may contain temperatures of different time periods of the same day. It can be represented by a vertex. Inside a vertex, a structure is used to hold which part of the data object is accessed, what kind of operations (read/write) it is and the time cost of accessing. For example, it is possible that all the temperatures are accessed, or only the temperatures of every two hours are accessed. Recording which part of the data object is accessed can improve the accuracy of prefetching, since an application may use partial data of an object. For instance, it may read odd columns of data object A with odd rows of data object B. If this pattern is fixed, we can always try to prefetch the proper parts of data object A and B. The edges in the graph represent the paths of traversing the data objects by the application. Edge $V_1 \rightarrow V_2$ indicates that the application visits V_2 after V_1. The weight of an edge is the time length between the visits of V_1 and V_2. Figure 5 shows a sample accumulation graph and Figure 6 shows a close view of a single vertex of the graph.

Each time the application is run, new information is added to the graph, which is the process of accumulating. If the application is run with the same I/O behaviors, the accumulation graph remains unchanged. If a divergence occurs, a branch is added to the graph. When a new I/O operation is detected, we always try to merge it with existing paths. For example, in Figure 5, the application diverges at V_2 and then the paths merge at V_5.

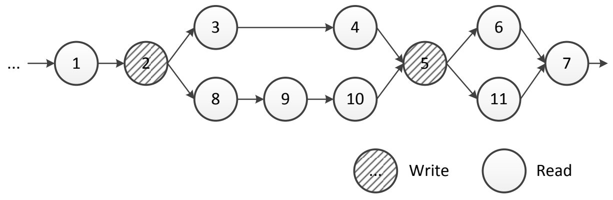


Fig. 5. A sample accumulation graph.

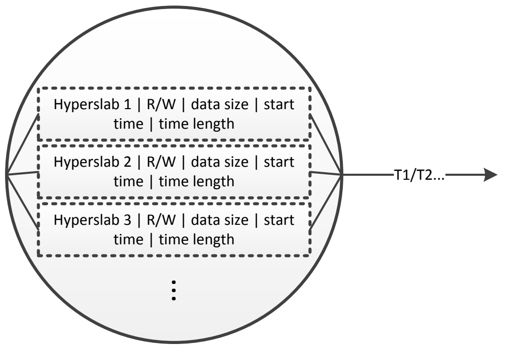

Hyperslab 1 | R/W | data size | start time | time length
Hyperslab 2 | R/W | data size | start time | time length
Hyperslab 3 | R/W | data size | start time | time length
T1/T2...

Fig. 6. A single vertex in the accumulation graph.

V. Design and Implementation

A. PnetCDF as the framework

NetCDF stands for Network Common Data Format. It is a set of formats, libraries and tools that is widely used by scientific computing communities [29]. Parallel NetCDF (PnetCDF) is a parallel version of NetCDF, which supports data parallelism. The data elements in PnetCDF are dimension, variable, and attribute. They are usually assigned meaningful names to reflect their nature. For example, a variable $temperature$ may have two dimensions named city and time. This variable may record the temperatures of different cities over time. PnetCDF provides clear labels for its data objects and thus has clear semantics boundaries, which can be utilized for accumulating high-level application-specific knowledge. The methodologies we used in this paper can also be applied to Parallel HDF5.

As illustrated in Figure 2, PnetCDF actually uses MPI-IO to conduct I/O operations. It couples closely with MPI and uses MPI datatype to organize its data.

B. Accumulation: tracing, analyzing and storing

In order to get the application and machine specific information, PnetCDF is modified to support tracing I/O behaviors of the application, analyzing the traces and save results in the knowledge repository for future uses. All of the fundamentals are integrated into PnetCDF to make it transparent to applications and the underlying file systems.

To record the high-level I/O behaviors and keep the PnetCDF interfaces unchanged, we added prefix 'P' to all the API names (including the definition and caller) and keep declaration names in header files unchanged. We then re-implemented the interface functions to incorporate the functionalities we need. For example, we changed $ncmpi\_get\_vars()$ to $Pncmpi\_get\_vars()$ in $getput\_vars.c$ where the function is defined and elsewhere it is called, except the declaration in pnetcdf.h where the interface to the outside world is declared. Then we re-define the function $ncmpi\_get\_vars()$ which wraps the $Pncmpi\_get\_vars()$ and several functionalities we need. In summary, we added a layer between applications and the original PnetCDF to carry out our missions.

In order to process and store application-specific knowledge, we need to identify different applications. To recognize an application in PnetCDF, the users have two options. The first is to set a macro $ACCUM\_APP\_NAME$ when compiling the application. For example, for the applications that are developed with C and use Autoconf [30], a widely used configuration tool in Unix/Linux community, we can set the macro by adding "$CFLAGS = -DACCUM\_APP\_NAME = myapp01$" as its parameter. We attach this name to the application as an ID so that PnetCDF can recognize it. If the source code of the application changes, the user can assign it with a new name to identify new patterns of the application. The second way is to set a global environment variable, $CURRENT\_ACCUM\_APP\_NAME$. When an application is executed after this environment variable is set, the application's name ($ACCUM\_APP\_NAME = myapp01$) in PnetCDF will be overwritten by $CURRENT\_ACCUM\_APP\_NAME$ and the application has a new ID. The second solution provides a supplement to the first one by making KNOWAC prefetch system more flexible. By using it, we can actually set up several knowledge repositories/profiles for an application or share the same knowledge repository/profile among several applications. It provides a way to improve a single profile and to use a profile to improve related applications. For example, a project may have several tools that all have similar I/O patterns. In this case, all of them can share an ID in the knowledge repository.

KNOWAC repository is stored in SQLite database [31]. SQLite is a lightweight library that implements a SQL engine. It stores the entire database into a single cross-platform file. This makes the knowledge very portable since we can move the database file around and use it on different platforms for prefetching.

An analyzer is used to examine I/O behaviors and accumulate the knowledge to the repository. The analysis is on-line. Since the collected I/O behaviors are high-level, the metadata is usually not large. After we uniquely identify an application, the first thing is to determine whether we already have its profile. If not, we record the I/O behaviors and build a new accumulation graph. If the application has been run with the KNOWAC-enabled PnetCDF, we check the current data path with the data graph of the application and schedule prefetch tasks.

C. Cooperation between main and helper thread

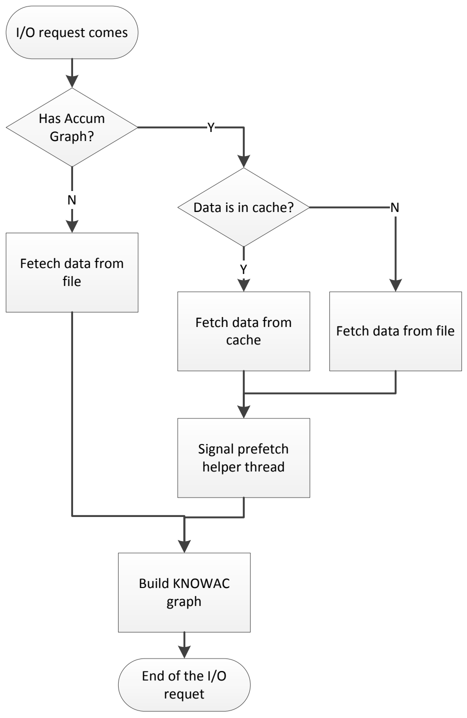


Fig. 7. Flowchar of the main thread

In KNOWAC prefetch system, the main thread of the application remains the same. However, the PnetCDF APIs that are called by main thread are prefetch-enabled. A prefetch helper thread is used to conduct prefetching. Figure 7 is the flowchart of the main thread when the application requests I/O operations. Whenever the application invokes an I/O interface, internally it first checks if there is an accumulation graph for this application in the knowledge repository. If not, it calls original I/O routines and builds a new accumulation graph for this application. Otherwise, prefetching is enabled and the

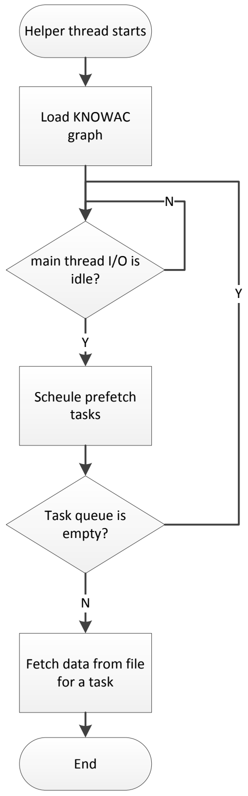


Fig. 8. Flowchar of the prefetch helper thread

system checks if the data needed are in cache. It then fetches the data from either cache or file. After this I/O operation is done, main thread informs the prefetch helper thread the status of the last I/O operation.

To use the idle I/O time while not disturbing the computation, a helper thread is spawned to conduct prefetching. Figure 8 shows the control flow of it. The helper thread loads KNOWAC accumulation graphs for a particular application. After that, when there is no I/O operation in main thread, it analyzes the I/O behaviors of the main thread and schedules a prefetch task based on analysis results. Whenever there are any prefetch tasks in the queue and the main thread I/O is idle, the system fetches data from the file to the cache.

D. Predict and fetch

To prefetch effectively, we need to decide what and when to prefetch. Prefetching wrong data is a waste of CPU time and I/O bandwidth. Prefetching at a wrong time could have a negative impact on other I/O operations. To prefetch effectively, we trigger prefetching during I/O idle time (computation phases, for example) with knowledge of the application and machine-specific information. We call the prefetching of a data variable a prefetching task.

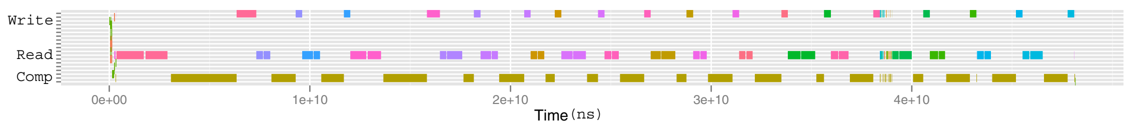
To predict future accesses, we first match current I/O behaviors with patterns in accumulation graphs. This process is started at the end of each main thread I/O operation. The main thread signals the helper thread after an I/O operation. Then the helper thread performs matching by inspecting the I/O behaviors of the main thread. If the main thread has not done any I/O, the application is at the starting point of the graph. If it has a sequence of I/O behaviors, then the sequence is searched in the accumulation graph. When a new I/O operation occurs, we check whether it follows the path we found last time. If not, we start matching again. In the case where no match can be found, we cut out the oldest I/O operation from the I/O sequence and do the match again. When multiple matches are found, the system extends the sequence to include an older operation and starts matching again. If the system reaches a point when no older operation can be added and we still have multiple matches, it simply passes it to the next stage and let the prediction component make a proper decision.

Once the system finds a match of the application's I/O behaviors in the accumulation graph, it can predict future I/O behaviors by following the path. If there are multiple future I/O operations, the system picks the one that is visited most. If they are equally visited, the system picks one randomly. After the prediction, a scheduler, which is embedded in the helper thread, is started to fill the I/O idle time with prefetching tasks. The idle time is estimated based on previous experience, which is stored in the accumulation graph. Tasks are scheduled one by one. The number of tasks are constrained by the cache size and number of tasks allowed in cache. We always prefetch if there is enough cache. If the prediction is very accurate, the next needed data will be the prefetched data. The cache size as well as the number of variables allowed in cache can be set to a smaller value to limit the number of variables to be prefetched. If a path in a graph has many branches, we have the choice to prefetch variables of multiple branches, if memory space is available and I/O is not high. For example, in Figure 5, we may fetch both V_3 and V_8. This approach is especially useful for applications with many different short computation phases with many data dependencies in it.

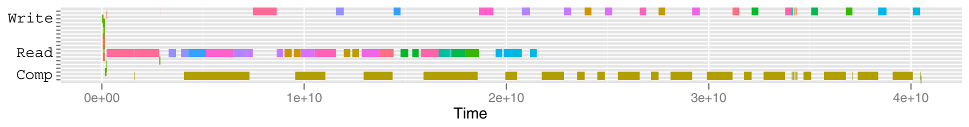
The number of branches in the accumulation graph influences the accuracy of prefetching prediction, unless we prefetch all the possible branches. To improve the accuracy and effectiveness of prefetching, users can manipulate the knowledge repository by using $CURRENT\_ACCUM\_APP\_NAME$ environmental variable. If users are familiar with their applications, they can tailor $CURRENT\_ACCUM\_APP\_NAME$ to avoid generating more branches. The extreme case is that if the application's behaviors never change, using the same $CURRENT\_ACCUM\_APP\_NAME$ would achieve best performance since prediction is always accurate. In cases like this, a little human intervention would make big difference. Ten seconds of setting up the environment variable in script could possibly gain performance improvements of hours or days.

VI. Evaluation

We used a real-world data-intensive application to verify the concept and design of the KNOWAC prefetching system. The experiments were conducted on a 64-node cluster. These



(a) Without KNOWAC prefetching



(b) With KNOWAC prefetching

Fig. 9. I/O behaviors of a typical *pgea* run. The colors indicate the logical meaning of the data.

nodes were Sun Fire X2200 servers with dual 2.3GHz Opteron quad-core processors, 8G memory, 250GB 7200RPM SATA hard drive and 100GB PCI-E OCZ Revodrive X2 SSD (read: up to 740 MB/s, write: up to 690 MB/s). Ethernet and Infiniband were both equipped on the cluster. The operating system was Ubuntu 9.04 (Linux kernel 2.6.28-11-server). MPI implementation was MPICH2-1.4.1p1 and the parallel file system we used was PVFS2 2.8.2 with stripe size of 64KB. We used 4 I/O servers if not specified.

A. *GCRM and Pagoda*

Global Cloud Resolving Model (GCRM) [32] is a model used to simulate global climate on the scale from two to four Km. Its data are of petascale. For four Km resolution, if Pagoda produces data every three hours for one simulated year, 1.4 PB of data will be produced [28]. Analyzing data of such scale requires great I/O capacity but current I/O systems can hardly meet the requirement. The GCRM data have explicit topology variables as many other scientific applications. The data are in the format of NetCDF and the data model is relatively fixed. The dimensions include time, cell, corner, edges and so forth. The variables, which are big arrays, include temperature, heat and so forth. Thanks to NetCDF, the names of dimensions, variables and attributes can be retrieved and used. The names can be considered as the data's logical information which can be utilized to get high-level patterns. Low-level libraries, such as MPI-IO and POSIX, do not have this property naturally, which makes it extremely hard to retrieve logical I/O behaviors from these low-level libraries.

Pagoda [28] stands for Parallel Analysis of Geoscience Data. It attempts to mitigate the I/O bottleneck of GCRM data analysis by data parallelism through PnetCDF. Pagoda is both a set of APIs and tools based on the APIs. Although it is designed for geodesic semi-structured NetCDF data in the first place, they can be used for regularly gridded data.

Pagoda includes several tools. *pgea* performs grid point averaging on the input files, with each file receiving an equal weight in the average. *pgea* can perform linear average as well as other operations, such as square average, max, min, rms, random rms. We use *pgea* to test KNOWAC. The I/O behaviors of the original *pgea* can be observed in Gantt chart in Figure 9(a). In each phase, it first reads variables from the input files (two files in this case), conducts the computation and then writes the variable to a new file. In Figure 9(b), the I/O behaviors of a KNOWAC-prefetch-enabled *pgea* are shown. We can see that most of the variables are prefetched before they are used. The prefetch I/O is overlapped with computation. 16% of the execution time is reduced in this case.

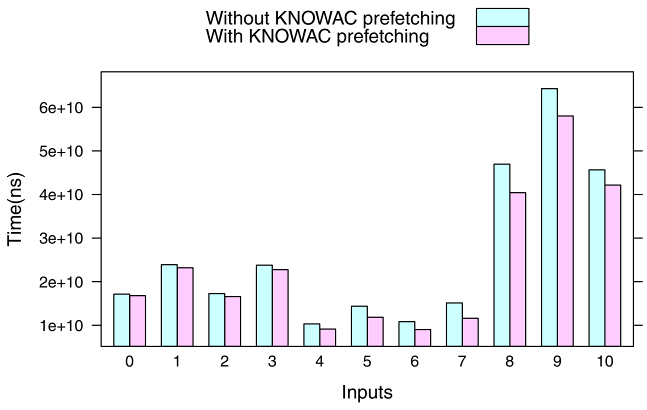


Fig. 10. Execution time of inputs with different sizes and formats

B. *Execution Time Improvement*

To evaluate the improvement on execution time and test whether the knowledge can be used for different data inputs, *pgea* is run with same parameters but different input. Re-running an application with different inputs is a common scenario in scientific computing. For example, some tools are constantly used with the parameters, to pre-process formatted data produced by instruments/simulations. Figure 10 shows the overall execution time of *pgea*. We can see improvements on all inputs. The variance of improvements depends on many factors, such as the amount of computation, the sequence of data reads, etc.

Figure 11 shows the improvement of prefetching with different operations on data. Since we try to overlap computation with I/O, the time of computation is critical. Because if there is no computation (i.e. application has pure I/O), no overlap

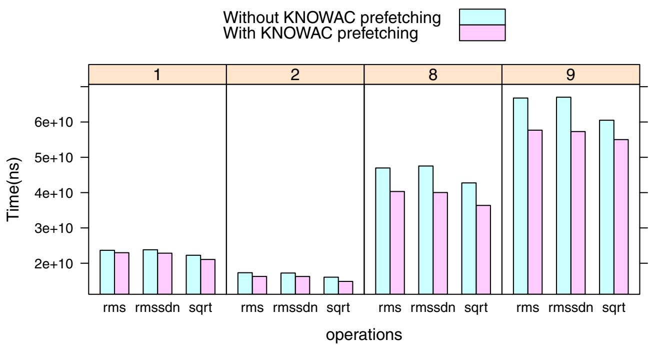


Fig. 11. Execution time with different computation operations.

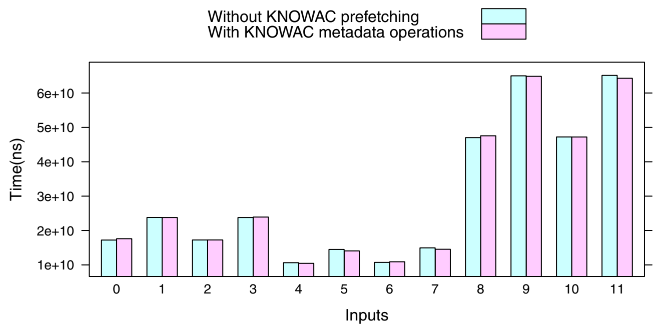


Fig. 13. Execution time with the influences of the prefetch metadata management and helper thread

can be done to shorten the overall time. If the computation time is too short, KNOWAC will not schedule a prefetching task. Because, in that case, there is not much overlap and the prefething I/O may interfere in the original I/O. If there is more time spent on computing, the overlap of computation and I/O can be larger. As we show in the results, applications with intensive I/O and a fair amount of computation can gain considerable improvements.

C. Scalability

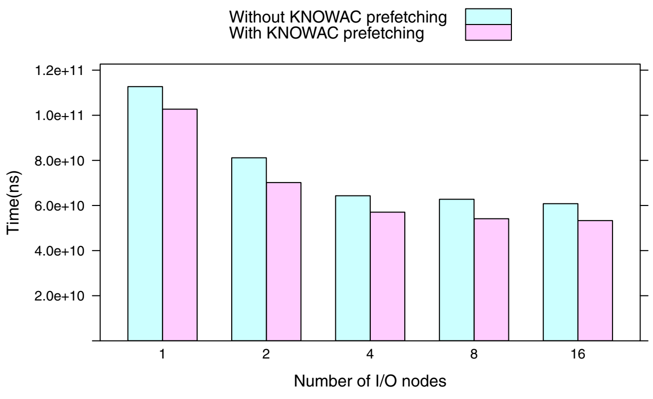


Fig. 12. Scalability of KNOWAC prefetching system

The scalability is tested based on the fixed-size scalability [33]. The number of I/O nodes is increased, but the input data remain the same. Figure 12 shows that, prefethcing can improve the performance at different scales. This test shows when the underlying I/O or file systems becomes faster, the overall execution time will be reduced accordingly. However, prefetching is still important.

D. Overhead

To measure the time cost of the metadata operations and extra costs like thread spawning, we remove the prefetching I/O calls and keep everything else the same in the prefetching system. In other words, the operations of the KNOWAC graph and the prefetch helper thread are still there, but there is no I/O in the prefetch thread. In this case, we can see how much overhead we introduce to the system.

Figure 13 shows the execution time of $pgea$ with different sets of inputs. We can see the variations of the execution time are small. The results indicate that the metadata management overhead of KNOWAC is ignorable. Since the system collects and analyzes data on a high-level, the metadata size is small and can be managed efficiently.

E. Performance on SSD

Solid State Disk(SSD) is a promising technology to bridge the gap between computation and I/O. KNOWAC prefetching is tested on SSD. Figure 14 shows that the KNOWAC prefetching works as well on SSD and the improvement is significant. By inspecting the results closely, we also see that the execution time standard deviations of system with SSD are smaller than that with HDD, showing that systems with SSD are more stable.

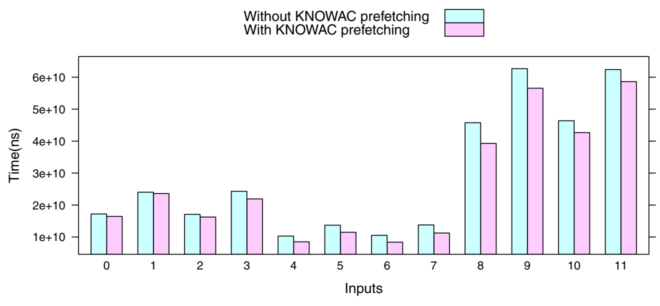


Fig. 14. Execution time of inputs with SSD

VII. CONCLUSION AND FUTURE WORK

I/O becomes the performance bottleneck of modern cluster computing, especially for data-intensive applications. However, conventionally, I/O has been considered as a peripheral, which is stateless and only with the ability to handle data in terms of bytes. Current I/O system is not application-aware and very hard to be optimized based on application I/O characteristics. In this study, we propose a stateful I/O system named KNOWAC that can accumulate knowledge of applications and machines and utilize the knowledge to prefetch data ahead of time. KNOWAC collects high-level I/O behaviors, reveals data access patterns and stores the knowledge of the application to a repository for future uses. KNOWAC prefetch system is implemented under the framework of PnetCDF and evaluated extensively with a real scientific application. It is practical and effective. Experimental results show that KNOWAC has a real potential in reducing I/O data access time.

Developing smart and application-aware I/O systems is a long-term task. KNOWAC is a first step to show that collecting and using high-level knowledge for I/O optimizations are possible, and has a promising future. We believe knowledge collected and analyzed by KNOWAC I/O system is not only applicable to prefetching, but also applicable to other I/O optimizations. In the future, we plan to further explore high-level I/O knowledge, develop runtime systems, and study new applications of the application-aware, smart I/O system approach.

ACKNOWLEDGMENT

The authors are thankful to Karen Schuchardt of Pacific Northwest National Laboratory, Quincey Koziol of HDF Group, Rob Ross of Argonne National Laboratory for their constructive and thoughtful suggestions toward this study. This research was supported in part by National Science Foundation under NSF grant CCF-0621435 and CCF-0937877.

REFERENCES

[1] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale i/o workloads," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–10.

[2] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, p. 8, 2011.

[3] A. Hey, S. Tansley, and K. Tolle, *The fourth paradigm: data-intensive scientific discovery*. Microsoft Research Redmond, WA, 2009.

[4] 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.

[5] P. Carns, W. Ligon III, R. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters," in *Proceedings of the 4th annual Linux Showcase & Conference-Volume 4*. USENIX Association, 2000, pp. 28–28.

[6] P. Braam, "Lustre file system: high-performance storage architecture and scalable cluster file system. Whiter Paper, Sun Microsystems," *Inc., Santa Clara, CA, USA*, 2007.

[7] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the First USENIX Conference on File and Storage Technologies*. Citeseer, 2002, pp. 231–244.

[8] J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netcdf: A high-performance scientific i/o interface," in *Supercomputing, 2003 ACM/IEEE Conference*. IEEE, 2003, pp. 39–39.

[9] F. Chang and T. Mowry, "Using speculative execution to automatically hide i/o latency," 2001.

[10] R. Patterson and G. Gibson, "Exposing i/o concurrency with informed prefetching," in *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*. IEEE, 1994, pp. 7–16.

[11] R. Patterson, G. Gibson, and M. Satyanarayanan, "A status report on research in transparent informed prefetching," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 2, pp. 21–34, 1993.

[12] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, *Informed prefetching and caching*. ACM, 1995, vol. 29, no. 5.

[13] F. Chang and G. Gibson, "Automatic i/o hint generation through speculative execution," *Operating systems review*, vol. 33, pp. 1–14, 1998.

[14] Y. Chen, S. Byna, X. Sun, R. Thakur, and W. Gropp, "Hiding i/o latency with pre-execution prefetching for parallel applications," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*. IEEE, 2008, pp. 1–10.

[15] S. Byna, Y. Chen, X. Sun, R. Thakur, and W. Gropp, "Parallel I/O prefetching using MPI file caching and I/O signatures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 44.

[16] J. Oly and D. 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.

[17] H. Lei and D. Duchamp, "An analytical approach to file prefetching," in *USENIX Annual Technical Conference*, 1997, pp. 275–288.

[18] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "Diskseen: exploiting disk layout and access history to enhance i/o prefetch," in *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2007, pp. 1–14.

[19] M. Kallahalla and P. Varman, "Optimal prefetching and caching for parallel i/o sytems," in *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2001, pp. 219–228.

[20] A. Nisar, W. Liao, and A. Choudhary, "Scaling parallel i/o performance through i/o delegate and caching system," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*. IEEE, 2008, pp. 1–12.

[21] K. Coloma, A. Choudhary, and W. Liao, "Dache: Direct access cache system for parallel i/o," in *In to appear in Proceedings of the 2005 International Supercomputer Conference*. Citeseer, 2005.

[22] K. Iskra, J. Romein, K. Yoshii, and P. Beckman, "Zoid: I/o-forwarding infrastructure for petascale architectures," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 153–162.

[23] IBM Blue Gene/Q supercomputer delivers petascale computing for high-performance computing applications. [Online]. Available: http://www-01.ibm.com/common/ssi/rep_ca/8/897/ENUS112-028/ENUS112-028.PDF

[24] J. He, J. Kowalkowski, M. Paterno, D. Holmgren, J. Simone, and X.-H. Sun, "Layout-aware scientific computing: a case study using milc," in *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems in conjunction with ACM/IEEE SuperComputing 2011*, 2011.

[25] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[26] Montage website. [Online]. Available: http://montage.ipac.caltech.edu/

[27] HDF-EOS website. [Online]. Available: http://www.hdfeos.org/

[28] Pagoda website. [Online]. Available: https://svn.pnl.gov/gcrm/wiki/Pagoda

[29] NetCDF website. [Online]. Available: http://www.unidata.ucar.edu/software/netcdf/

[30] Autoconf website. [Online]. Available: http://www.gnu.org/software/autoconf/

[31] SQLite website. [Online]. Available: http://www.sqlite.org/

[32] GCRM website. [Online]. Available: https://svn.pnl.gov/gcrm/

[33] X. Sun and L. Ni, "Another view on parallel speedup," in *Supercomputing'90. Proceedings of*. IEEE, 1990, pp. 324–333.