Dynamic process migration based on block access patterns occurring in storage servers

Jianwei Liao, François Trahay, Guoqiang Xiao

▶ **To cite this version:**

Jianwei Liao, François Trahay, Guoqiang Xiao. Dynamic process migration based on block access patterns occurring in storage servers. ACM Transactions on Architecture and Code Optimization, 2016, 13 (2), pp.20. 10.1145/2899002 . hal-01347983

HAL Id: hal-01347983
https://hal.science/hal-01347983v1

Submitted on 22 Jul 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Dynamic Process Migration based on Block Access Patterns occurred in Storage Servers

JIANWEI LIAO
Southwest University, China
FRANCOIS TRAHAY,
CNRS Samovar
Télécom SudParis
Universit Paris-Saclay
GUOQIANG XIAO
Southwest University, China

***Abstract*—An emerging trend in developing large and complex applications on today's high performance computers is to couple independent components into a comprehensive application. The components may employ the global file system to exchange their data when executing the application. In order to reduce the time required for I/O data exchange and data transfer in the coupled systems or other applications, this paper proposes a dynamic process migration mechanism on the basis of block access pattern similarity for utilizing the local file cache to exchange the data. We first introduce the scheme of block access counting diagram, to profile the process access pattern during a time period on the storage server. Next, we propose an algorithm that compare the access patterns of processes running on different computing nodes.. At last, processes are migrated in order to group processes with similar access patterns. Consequently, the processes on the computing node can exchange their data by accessing the local file cache, instead of the global file system.**

The experimental results show that the proposed process migration mechanism can reduce the execution time required by the application because of the shorter I/O time, as well as yield attractive I/O throughput. In summary, this dynamic process migration technique can work fairly well for distributed applications whose data dependency rely on distributed file systems.

I. Introduction

High performance clusters and other high performance distributed computer systems, which consist of computing nodes for executing processes and storage nodes for providing I/O services, are widely used to solve large scientific applications [Vecchiola 2009]. However, due to the increase of application scale and of the data size, one of the most critical challenges to be addressed is how to overcome the data access bottleneck [Dongarra 2011].

Specifically, the noticeable issue arises from the study area of data I/O in high performance computing: the I/O transfer time needed for exchanging data among processes belonging to the same application depends on both the affinity of the processes and the locations of processes [Jeannot 2014]. That is to say, if two processes are located on the same node, both of them can exchange data by using the local file cache. As a consequence, the time required for exchanging data using the local file can be considerably reduced compared to exchanging the data via the parallel file system.

In other words, preserving data locality by placing the computation tasks close to their required data is crucial for performance in large clusters, because the slow data rate on the disk and the network bisection bandwidth become a bottleneck [Dean 2008]. In order to reduce the cost of inter-process communication (i.e. MPI communication [MPICH2]), many studies have focused on binding processes to the proper computing nodes [Chen 2006,Mercier 2011]. These mechanisms make sense if and only if the binding process knows the communication pattern of the application (i.e., which process communicate with which process).

However, many problems in science and engineering can be solved by coupling interacting models, which may result in several independent/semi-independent models for handling the different steps. For example, in modeling long-term global climate, the Community Climate System Model (CCSM) includes an atmosphere model, an ocean model, a sea-ice model and a land-surface model [Hack 2006]. On the other side, these models have been separately developed by different researchers, thus, they may present distinct behaviors deriving from their interdisciplinary essence and from their computation flow in the algorithms [Larson 2005,Valcke 2012]. As a result, the processes belonging to the different models do not have the information about where to send/get their output/input data. For this reason, this kind of application commonly leverages the global file system for exchanging the data. However, disk-based data I/O operations not only slow down the execution of the application, but also place pressure to the file system.

This paper proposes a mechanism that analyzes the I/O transfers of a distributed application and dynamically migrates processes between computing nodes in order to improve the locality of I/O accesses. Therefore, instead of using the distributed file system, processes can exchange data via the local file cache on the computing node, which reduce the cost of data transfers. In this mechanism, we first employ the newly proposed scheme of block access counting diagram to profile the access pattern on the storage server for each process; then we compute the access pattern similarity of two processes. Depending on their similarity, some of the processes may then

be dynamically migrated to other computing nodes in order to improve the locality of I/O accesses and to yield a better system performance. In short, this paper makes the following two contributions:

1) On the storage server side, we propose the techniques of block access counting diagram and *N*-compressed access counting diagram, to classify block access patterns in a certain period with a per-process granularity. The access patterns are then compared in order to estimate the pattern similarity of two processes.
2) We also propose a novel scheme to perform process migration based on the similarity of access patterns. This migration strategy groups processes with similar access patterns together so as to improve the data locality and thus maximize the usage of the local file cache. As a consequence, besides accelerating the I/O operations, the data rate of the I/O subsytem is greatly improved.

The rest of the paper is structured as follows: the related work regarding process migration is described in Section II. The design and implementation details of this newly proposed process migration mechanism are illustrated in Section III. Section IV introduces the evaluation methodology and discusses experimental results. At last, we make concluding remarks in Section V.

II. Related work

Many previous work have focused on process migration, block access pattern-based I/O optimization, and the coupling tools for independent models:

Dynamic Process Migration: The mechanism of dynamic process migration by using the checkpoint/restart facility is widely used for a variety of purposes: ①. *Dynamic processes migration for balancing workloads.* C. Du and X. Sun et al. proposed a dynamic scheduling mechanism that takes migration cost and other conventional influential factors into account, to boost system performance in a shared, heterogeneous environment [Du 2007]. L. Pilla and C. Ribeiro et al. have introduced a novel hierarchical load balancing mechanism by adopting dynamic process migration, to enhance the performance of applications on parallel multi-core systems [Pilla 2012]. ②. *Dynamic processes migration for achieving fault tolerance.* Checkpoint/restart is the most popular technique in high performance computing to offer high availability and reliability. It is simple and effective enough in situations where the failures do not occur frequently. The fundamental principle of checkpoint/restart is to periodically save the state of the target process. If, for some reason, the process crashes, the most recent checkpointed state is read and the process execution restarts from there [Liao 2012a]. Although the idea of checkpoint/restart is quite straightforward, it adopts many variants in the specified application contexts [Meneses 2015,Dong 2011].

Especially, with the emerging of visualization technique [Williams 2012], certain virtual machine replication and migration approaches have been introduced. L. Cheng and C. Wang have proposed a new I/O virtualization approach called *vBalance* to substantially boost the I/O performance for Symmetric MultiProcessing (SMP) virtual machines [Cheng 2012] . This approach is a cross-layer solution to speedup I/O operations by migrating interrupts from a preempted vCPU to a running one. Through migrating operating systems running services with liveness constraints in a virtual machine, it is possible to rapidly move the interactive workloads within clusters and data centers, and then reach the targets of load balancing and fault tolerance [Clark 2005].

In addition, in order to enhance the communication performance among the MPI processes in multi-core clusters, E. Jeannot and G. Mercier et al. have proposed algorithms that place MPI processes on computing nodes according to their communication patterns. That is to say the processes that communicate a lot with each other are allocated on the same or nearby nodes. The overhead of inter process MPI communication is thus greatly decreased [Mercier 2011,Jeannot 2014]. P. Michaud et al. have proposed a thread migration mechanism, which is able to maximize system performance for ensuring the necessary thread migrations on the basis of a temperature constraint [Michaud 2007]. Q. Chen and M. Guo have presented an adaptive scheme of task scheduling on the basis of workloads, for asymmetric multicore architectures [Chen 2014].

Active Storage Systems & Other Intelligent Storage Systems: The active storage system is aimed at I/O bound applications that involve fundamentally independent data sets [Piernas 2010]. It takes advantage of the underutilized computing resources in the storage servers, and supports migrating the processes from computing nodes to the proper storage nodes [Piernas 2007,Xie 2015]. As a result, the processes can be executed on the storage nodes by using a large amount of data without any data transfers between the computing nodes and storage nodes, for achieving better system performance. However, the migrated processes are likely to use certain restricted kernel functions offered by the computing nodes in some cases, because of resource limitations on storage nodes, it might be impossible to execute this kind of tasks on the storage nodes. Furthermore, direct access to the storage nodes is not generally allowed in real-world production environments for security reasons [Zheng 2013].

To offer intelligent storage services for different I/O intensive workflows, L. Costa et al. have presented a mechanism to enable making runtime system configuration decisions including chunk size and caching policies, in a storage system, to speedup I/O processing in the case of specified application context [Costa 2014]. Z. Gong have introduced a high-performance parallel I/O middleware for large-scale HPC applications, to yield user-transparent layout optimization for scientific workflows, on the basis of access patterns of the applications [Gong 2013].

Block Access Patterns-based I/O Optimization: In order to yield better I/O performance, there are also substantive optimization strategies that employ the information about block access patterns on the storage nodes or the local file server. Z. Li and Y. Zhou et al. proposed a data mining approach

called *C-miner* to explore block correlations in the file server on a local machine. So that the file system can make use of the discovered block correlations for guiding I/O optimization strategies, such as data prefetching or data movement [Li 2004]. Similarly, S. Jiang et al. have implemented *DiskSeen*, which employs a frequent sequence-based pattern modeling technique to classify block access pattern, and both temporal and spatial correlations of block access events have been taken into account, for improving the sequentiality of disk accesses and overall prefetching performance [Ding 2007,Jiang 2013]. J. He et al. have proposed file re-organization on the basis of logical I/O patterns, which occur in the computing nodes, for expediting the I/Os [He 2013].

Moreover, after knowing the fact of the block access events occurred on the storage servers have certain regularities,

Moreover, we have proposed and implemented a server-side prefetching mechanism in distributed file systems in our previous work [Liao 2015] that takes into account the regularity of the block access events that occur on storage servers. This I/O optimization scheme utilizes the block access patterns to direct prefetching data on the storage nodes. Then the prefetched data is proactively forwarded to the client node, to speed up the execution of the applications running on the resource-limited computing nodes (i.e. client nodes), and to enhance I/O data throughput.

Coupling mechanisms for separated models or applications. The Model Coupling Toolkit (*MCT*) [Larson 2005] is a library providing routines and datatypes for creating a coupled system, and mainly used in Community Climate System Model (*CCSM*). Hereafter, S. Valcke et al. [Valcke 2012] have summarized major coupling technologies used in Earth System Modeling, and their paper shows common features of the existing coupling approaches including the functionality to communicate and re-grid data. Moreover, the *OASIS coupler* is another typical related work, which is able to process synchronized exchanges of coupling information generated by different components of the climate system, and the separate coupler mediates communication among the components [Valcke 2006]. However, these coupling tools require to modify the source codes of applications, to use their specific interfaces. That is to say, they cannot keep the transparency to the application users, and fail to offer a general solution to all cases.

III. Design and Implementation

We propose a novel mechanism that dynamically migrates processes from one computing node to another, to effectively use the local file cache on the computing node. The decision to migrate a process to another computing node is done according to the similarity of the block I/O access patterns of the computing processes.

Figure 1 gives an overview of the dynamic process migration mechanism when the processes that run on different computing nodes have heavy I/O data dependency. The storage servers regularly compare the I/O access patterns of client computing processes. If two processes have similar access

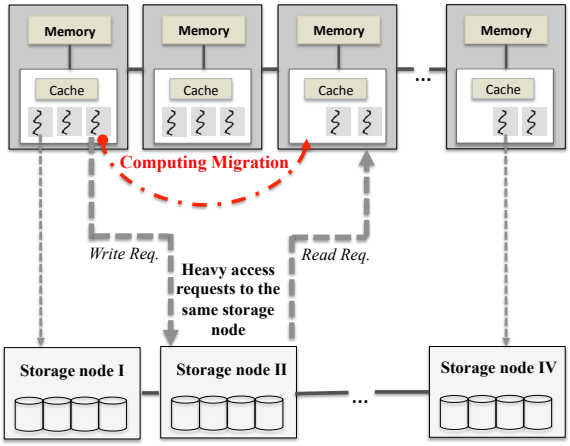


Fig. 1. The illustration of dynamic process migration on the basis of data dependency (note: data dependency can be represented with block access pattern similarity).

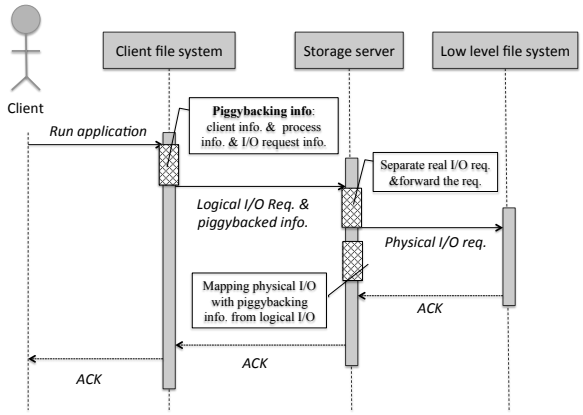


Fig. 2. Piggybacking mechanism for constructing mapping relationship between the logical I/O requests on client file systems and the physical I/O requests on storage servers with per-process granularity [Liao 2015].

patterns, the storage server triggers the migration of one process so that it is placed nead the process that has a similar access pattern. By improving the data locality, some of the processes I/O requests can be processed at the local file cache, which reduce the cost of I/O data transfer.

To put this mechanism to work, we propose a scheme of *Access Counting Diagram* to profile the block access patterns of processes, which are used to estimate the pattern similarity. After that, a checkpoint/restart mechanism is employed to migrate a process so that the processes that have similar access patterns share the same computing node.

A. Classifying access patterns

We designed a mechanism for caracterizing the I/O access pattern of processes. This mechanism runs on the storage nodes that receive requests for blocks of data from clients

file systems running on multiple nodes. Since a client file system can be solicited by multiple processes, we employ a piggybacking mechanism that has been implemented in our previous work [Liao 2015] in order to caracterize access patterns on a per-process basis. The sequences of I/O requests of each process are then stored as an *Access Counting Diagram* for further analysis.

1) Piggybacking mechanism: Figure 2 illustrates the piggyback mechanism, in which the client file system is responsible for keeping extra information about the process and client file system. After that, it piggybacks the extra information with the relevant client I/O request, and then sends them to the corresponding storage server. On the other hand, the storage server parses the request to separate the piggybacked information and the real I/O request from the client request. In addition to forwarding the I/O request to the low-level file system, the storage server also records the disk I/O access event with the piggybacked information about the process identifier etc. As a result, the storage server makes a record for each block access event along with the information piggybacked by the corresponding client I/O request, and then it is possible to profile block access patterns belonging to a specific process[1].

2) Profiling block access patterns: To model the block access pattern for a specific process and then benefit to matching patterns exactly and quickly, we have propose an approach of space-time diagram, called *Access Counting Diagram*. Both temporal and spatial correlations of block access events have been taken into account in this approach. Figure 3(a) illustrates the basic idea of the proposed space-time diagram for profiling block access patterns. In the figure, the X-axis indicates the time periods split with a fixed interval; and the Y-axis represents the range of accessed blocks. All the access tracks that occurred in the time window (*t0* to *t6* in the figure) are represented as points in the figure. Figure 3(b) shows how to count the events in each grid, and then a two-dimension array, i.e. a matrix, is used to represent the access pattern relevant to the process in the designated time period, i.e., time window.

Because a process may access a huge amount of blocks, the size of the pattern array may become quite large. To reduce the size of the pattern array for quickly matching the block access patterns, we propose the *N*-compressed counting scheme, which can coarsen $N \times N$ grids into a big one, and sum the values in the grids. Figure 3(c) shows an example of the 2-compressed counting conversion of Figure 3(b). In this case, the number of elements in the matrix is reduced from 36 to 9. As a result, the time needed for matching patterns is decreased to a great extent, though it may introduce less accuracy in pattern matching. Section IV discuss the impact of scaling the value of *N* in the compressed counting scheme.

B. Computing access pattern similarity

In order to estimate the degree of data dependency of two processes running on different computing nodes, we introduce the notion of pattern similarity. Equation 1 summarizes the algorithm that computes the similarity score of two block access patterns. The patterns are represented as two matrices having m $\times$ n elements (i.e. *m* time intervals and *n* block ranges.). A similarity score close to 1.0 indicates a high similarity, and that the two processes frequently access the same blocks of data.

$$\mu = f(A_{m\times n}, B_{m\times n}) = 1 - \frac{\sum_{i=0}^{m-1}(\sum_{j=0}^{n-1}\frac{|A_{i,j}-B_{i,j}|}{Max_{event}})}{m \times n} \quad (1)$$

where $A_{m\times n}$ and $B_{m\times n}$ mean the two involved matrices; $|A_{i,j} - B_{i,j}|$ represents the absolute difference of two integers, i.e. $A_{i,j}$ and $B_{i,j}$, which are two corresponding elements in the different matrices; Max_{event} is a pre-defined integer value, and set as the maximum number of block access events in the grid element by default.

As mentioned before, we keep the original access counting diagram and the compressed access counting diagram to compare the access pattern of processes. Therefore, the comparison algorithm first computes the similarity scores of the coarse-grain diagrams in order to promptly detect and discard the processes that have dissimilar access patterns. The fine-grain similarity is then computed only if the coarse-grain similarity is greater than a pre-defined threshold. The default value of *similarity threshold* is fixed as *0.90*. To put it from another angle, there are two steps in calculating the access pattern similarity:

- *Step 1:* the algorithm computes the similarity between two compressed access counting matrices. If the value of μ is larger than a pre-defined threshold, i.e., *similarity threshold*, it continues to *Step 2* for a fine-grained check. Otherwise, it stops and returns *false*, which means the result of "no similarity".
- *Step 2:* the algorithm conducts the pattern similarity computation between two original access counting matrices. When the value of μ is greater than the pre-defined threshold, it stops and returns *true* that indicates the result of "having similarity"; otherwise, it returns *false* that stands for "no similarity".

C. Dynamic process migration

This section discusses the details about dynamic process migration by using the MPI communication facility, as well as the checkpoint/restart mechanism.

1) Triggering migration: When a storage server detects two processes with similar block access patterns, it triggers the migration of a process in order to group the processes on the same computing node. Consequently, the two processes can use the local file cache on the client file system for exchanging intermediate data, rather than pulling/pushing the data from/to the storage nodes.

For the purpose of selecting a process to be migrated and choosing the destination node for the migrated process, the storage server first summaries the patterns' scales, according to

[1]Note that the piggybacking mechanism is not a new idea. It has been proposed in our previous work [Liao 2015], but it is a key technique to design and implement the newly proposed mechanism.

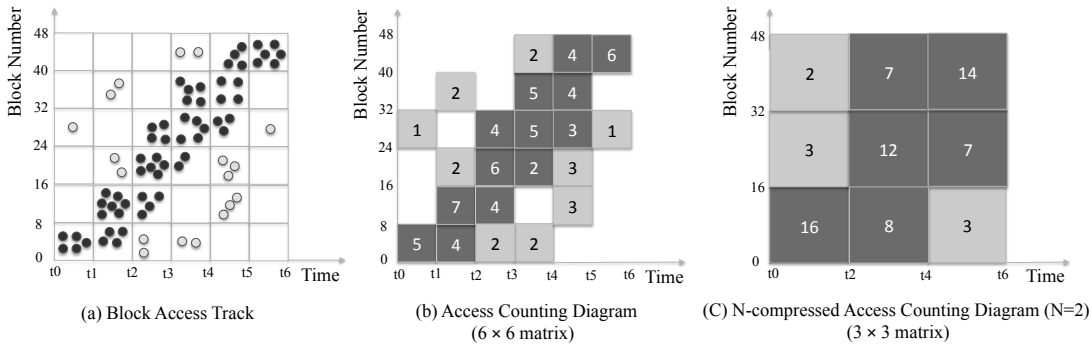


Fig. 3. Classifying block access patterns by using the scheme of space-time diagram. (a) forming a basic grid of space-time diagram to express the track of block access events; (b) counting the number of occurred block access in each grid element; (c) enlarging the size of grid element to reduce the number of grid elements.

Equation 2. These scales aggregate the number of I/O requests that were issued by a process over a period of time. Then, the storage server calculates the pattern similarity between the pattern having the largest scale and the patterns of the other processes running on other computing nodes according to Equation 1. If the storage server does not find a matched one, the process having the second-largest scale is selected and the server searches for a process that matches it. This algorithm is repeated until all the processes have been traversed. As a result, the found process(es) are migrated to the destination computing node(s).

$$\epsilon = f(A_{m\times n}) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} A_{i,j} \qquad (2)$$

where $A_{m\times n}$ represents the matrix of access pattern; and $A_{i,j}$ is the matrix element.

Certain cases may trigger the migration, for instance, periodical scanning conducted by the storage server tries to find the target processes to be migrated, this policy is currently used in our design. Besides, it is possible to allow the heavy-loaded computing nodes to issue process migration after resorting to storage servers for obtaining the information about the destination computing node. In summary, the storage servers are responsible for notifying both the source node and the destination node to be ready for process migration.

When a storage server decides to migrate a process from one computing node to another, the server notifies both the source node and the destination node, in order to trigger the migration process.

2) Migrating process: The checkpoint/restart facility is utilized to perform the process migration. Figure 4 shows the interaction between the two involved computing nodes for completing the migration.

On the source computing node, the client file system first sends a message for building a connection state with the destination node, so that both nodes are ready for conducting the process migration. Then, the client file system sends the consistent image of the targeted processes after certain synchronization operations.

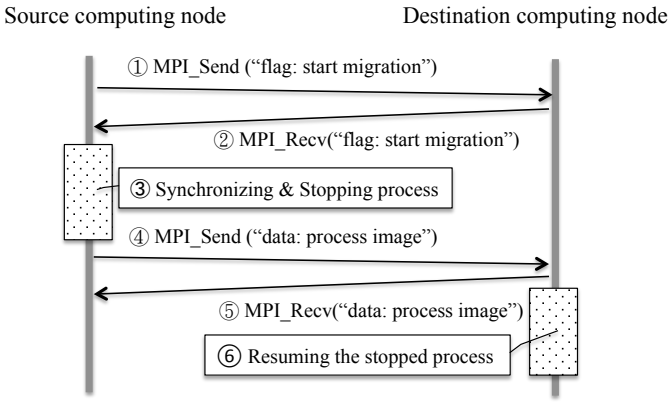


Fig. 4. The interaction between the source computing node and the destination computing node when performing a task of dynamic process migration.

On the destination computing node, the client file system restarts the stopped process when it has received the image of the targeted process. Consequently, the migrated process can continue running on the destination computing node, and is able to use the local file cache on the node for exchanging data with other processes running on the same node, which have the similar access pattern.

Note that the migration requires that the destination node has enough memory for hosting the migrated process. Thus, we configured the file system to abandon the migration when the size of the checkpointed state is bigger than one fourth of the memory capacity.

D. Implementation

Regarding the functionality of process migration, we have integrated the MPICH2 library [MPICH2] with the transparent kernel-level checkpoint/restart library implemented in our previous work [Liao 2012a] after certain modifications on the checkpoint/restart library, according to the idea presented in [Wang 2008]. Thus, we can save the consistent state of the process running on the source node, and transfer the

TABLE I
SPECIFICATION OF NODES ON THE CLUSTER I AND CLUSTER II

	Cluster I	**Cluster II**
CPU	4xIntel(R) E5410 2.33G	Intel(R) E5800 3.20G
Memory	1x4GB 1066MHz/DDR3	4GB DDR3-SDRAM
Disk	6x114GB 7200rpm SATA	500GB 7200rpm SATA
Network	Intel 82598EB, 10GbE	10GbE
OS	Ubuntu 13.10	Debian 6.0.4
# of Nodes	5	16

process image to the destination computing node by using MPI communication facilities, in order to resume the execution of the migrated process.

Moreover, we have implemented the newly proposed mechanism in the *PARTE* file system, which has been implemented from scratch in C, and runs in the Linux environment [Liao 2012b]. The implementation has three modules running at the user level:

- *partmds running on a specified server node.* The module of active metadata server, which offers metadata services for client file systems and storage servers.
- *parteost running on the storage nodes.* The module of storage server that manages real file data. Moreover, it is in charge of profiling block access patterns, computing pattern similarity, and issuing the command of process migration periodically.
- *partecfs running on the computing nodes.* The module of client file system has been designed and implemented based on FUSE [FUSE]. The client file system supports caching and managing for the cached data by employing the least recently used (LRU) policy. Furthermore, the client file system calls the checkpoint/restart library to fulfill dynamic process migration, after receiving the commands from the storage servers.

IV. EXPERIMENTS AND EVALUATION

This section describes the experimental methodology for evaluating the proposed migration mechanism, and then presents the experimental results.

A. Experimental setup

1) Experimental platform: Two clusters were used for carrying out the experiments: the active metadata server and 4 storage servers were deployed on the 5 nodes of Cluster I, and all the client file systems were located on the 16 computing nodes of Cluster II. Both clusters are equipped with MPICH2-1.4.1, and connected with a 1 GigE Ethernet. Table I shows the specifications of the nodes of both clusters.

2) Comparison counterparts & benchmarks: Because there is no related work that migrates on-going processes on the basis of I/O workloads, we used the PARTE distributed file system with the following three configurations to run the benchmarks:

- *Pattern-based Migration*, which is the distributed file system equipped with the mechanism we propose. It supports dynamic process migration according to the block access patterns. At first, the processes are allocated to the computing nodes by following the fashion of round-robin. But, the processes can be migrated to other computing nodes during the execution.
  We run the *Pattern-based Migration* scheme with different time windows. Namely, $Time_{window}$ is configured as 256 block access events or 512 block access events, we respectively label them as *Pattern-based Migration+256*, which is used by default, and *Pattern-based Migration+512*.
- *Non-Migration*, which does not enable process migration. The processes are bound to the computing nodes according to the round-robin fashion, and they are never migrated to other nodes.
- *Optimal Static Placement*, which is similar to the *Non-Migration* scheme, as it does not support process migration. The processes are statically allocated to the computing nodes, on the basis of their data dependency, after analyzing the source codes of benchmarks [Mercier 2009]. In other words, the processes in a process-pair are mapped to the same computing nodes, to enable exchanging their data by using the local file cache.
  Note that this scheme is not a general solution, as it must understand the data dependency among the processes, and the placement of processes never change during the life cycle. That requires to analyze the source codes of applications, or that the application users offers the dependency information. This comparison counterpart is leveraged to demonstrate the gain/loss in a global scale resulted by our proposed *Pattern-based Migration* scheme. Because we have modified the source codes of *mpi-io test* to emulate varied data dependency among the processes, we use this scheme while executing the benchmark of *mpi-io test*.

Moreover, several benchmarks and applications were selected to run on the above-mentioned storage systems, for evaluating the applicability of the proposed process migration mechanism. All the selected applications are scheduled to continuously run three rounds.

- *mpi_io_test*, which is constructed on top of MPI I/O calls and used to gather timing information for reading from and writing to file(s) using a variety of I/O profile configurations [MPI-IO Test]. In this benchmark, each process writes a large chunk of data to a non-overlapping, non-interleaved region of a file and then reads it back.
  We used *mpi_io_test* $v1.00.021$, and made two modifications to the code. First, in order to emulate the different processes having data dependencies, we have slightly modified this benchmark: process number *N* reads the data flushed by process number $N + i$, in the ith round of execution. Namely, the data dependency varies from different round of execution. The second modification involved enabling asynchronous I/O interface. That means a process can read the data, when another corresponding

process has written a part of data, the read and write operations are performed in parallel.

- *Scalable Synthetic Compact Applications (SSCA)*, which is a benchmark suite consisting of six benchmarks, including the simulations of real-world applications, such as *Bioinformatics Optimal Pattern Matching*, *Graph Analysis*, and *Synthetic Aperture Radar Application*. Since we focus on I/O performance, only the source code of SSCA #3 (version 0.1) was used in our experiments, which has *Sensor-Processing* and *Knowledge-Formation* steps, and focuses on large block data transfers and memory accesses, and small I/O operations [HPCS].
- *SPEEDY-LETKF*, which is a typical real-world coupled system including the SPEEDY model and the LETKF model. To be specific, *SPEEDY* is a simplified AGCM (Atmospheric General Circulation Model) developed by F. Molteni [Molteni2003]. In this experiment, it is coupled with the Local Ensemble Transform Kalman Filter (*LETKF*), which is a state-of-the-art approach to data assimilation [Hunt 2007]. *LETKF* reads the output data from *SPEEDY*, and then carries out the data assimilation. After that, the output results of *LETKF* are read by *SPEEDY* as the input data for continuing the next cycle of simulation.

3) Parameter Settings: In the PARTE file system, the block size of the system is 64KB, and the size of local file cache on the computing node was configured to *16* MB (thus, the client file system can cache up to *256* blocks of data), for which the Least Recently Used (LRU) replacement policy was used to replace cache blocks. The default value of Max_{event} was the maximum value of counting numbers among all grids in the access counting diagram. We configure the size of grid as 32×8, which means each grid may have a *32*-block range and there are *8* time intervals in a pattern. The value of *N* in the compressed access counting diagram was set to *2* by default. For each process, we keep the latest 3 block access patterns which are used for calculating the pattern similarity. In each round of scan, 1 process is fixed to be migrated when the corresponding process running on another computing node has been found.

There are also several pre-defined values for computing the pattern similarity in the experiments, according to the size of local file cache. The number of block access events, i.e. the time window of $Time_{window}$, is relevant to the size of the local file cache, we recommend to set the number between the range of $[Size_{cache}/Size_{block}, 2 \times Size_{cache}/Size_{block}]$. In other words, the larger time windows indicates the accurate pattern similarity, and this can reduce the number of unnecessary migrations. However, it requires more time for estimating the similarity, and it might not perform process migrations, even both processes have data dependency in reality. The storage servers calculated pattern similarity for the processes while they have completed $2 \times Time_{window}$ block access events from the previous stage. The experiments of running *mpi_io_test*, and *SSCA #3* have evaluated the proposed mechanism having different values of $Time_{window}$.

Furthermore, in the case study of real-world application, we have also changed the value of pre-defined *similarity threshold*, the value of *N* for the compressed access counting scheme, and the number of block access events used for profiling the access pattern, respectively in the case study. For simplifying the illustration, we set the value of $Time_{window}$ as 256, and the impacts of different values for $Time_{window}$ have not been checked in the case study anymore. The experimental results have illustrated the effectiveness with different level of triggers, and relevant results are reported in this section, as well.

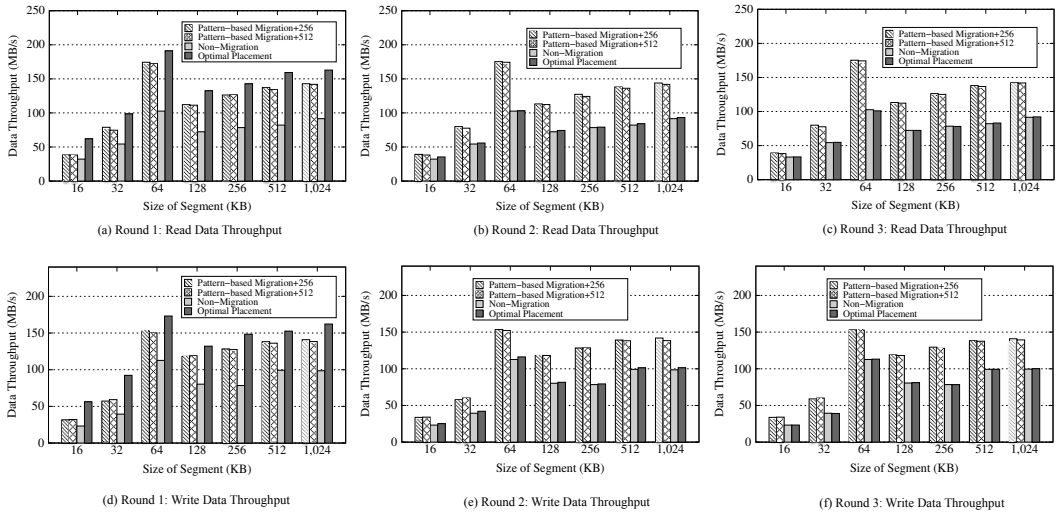
B. Experimental results

This section aims to disclose the effectiveness of the proposed mechanism, through running the selected benchmarks, as well as *SPEEDY-LETKF*, the real-world coupled application.

1) Results of mpi-io test: In our tests, we executed the benchmark with 64 MPI processes (4 MPI processes per computing node) that read or write one shared 16GB file. That is to say, each process p_i accesses the $(i + 16j)$th segment (with varied size) at call j, for $0 \leq i \leq 15$, generating a fully sequential access pattern. Since the processes read the pieces of data written by the different processes in the different rounds of execution, we present the experimental results in the different rounds, separately. Figures 5(a), (b), and (c) present the experimental results of read data throughput, and Figures 5(d), (e), and (f) report the write data throughput in the different rounds of execution, when the size of segment varies from each other, and the number of client nodes is fixed as 16 (i.e. 64 MPI processes). In the same way, Figures 6(a) - (f) show the I/O data rate, when the number of client nodes varies from 1 node to 16 nodes, but the size of segment is fixed to 64KB.

The experimental results show that the newly proposed mechanism improves the I/O data rate by $14.3 - 76.6\%$, in contrast to the *Non-Migration* scheme. Because the proposed mechanism of block access pattern-based process migration can indeed dynamically re-allocate the processes to the client nodes, on which there are processes with similar access patterns. Consequently, the processes are able to exchange their data by using the local file cache, instead of accessing the data through a storage server.

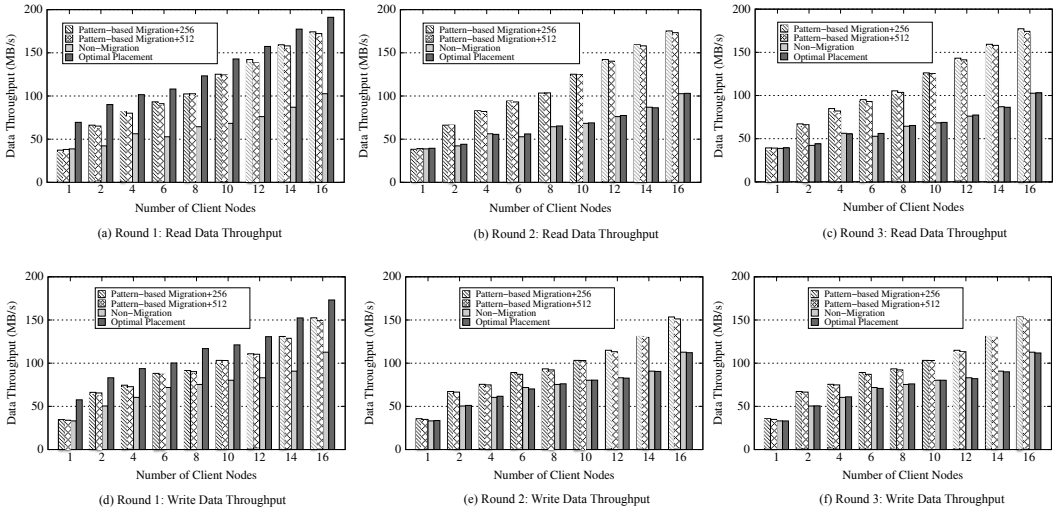
Further, the results reported in Figures 5 and 6 confirm that *Pattern-based Migration* works well when the access pattern changes during the execution. In the first round, the *Optimal Placement* scheme (that binds optimally the processes for the first round) outperforms the *Pattern-based Migration* scheme and *Non-Migration* (that place the processes in the round-robin fashion). This is because, for the first round, the *Optimal Placement* binds the processes according to their data dependency and that the processes can exchange their data through the local file cache. Although the *Pattern-based Migration* can dynamically migrate the processes for the same purpose, the migration causes an overhead that prevents this



(a) Round 1: Read Data Throughput (b) Round 2: Read Data Throughput (c) Round 3: Read Data Throughput

(d) Round 1: Write Data Throughput (e) Round 2: Write Data Throughput (f) Round 3: Write Data Throughput

Fig. 5. Data throughput obtained for the mpi-io benchmark with varying size of segment, when there are 16 client nodes. (a), (b), (c): Read data throughput in the different rounds of execution; (d), (e), (f): Write data throughput in the different rounds of execution.



(a) Round 1: Read Data Throughput (b) Round 2: Read Data Throughput (c) Round 3: Read Data Throughput

(d) Round 1: Write Data Throughput (e) Round 2: Write Data Throughput (f) Round 3: Write Data Throughput

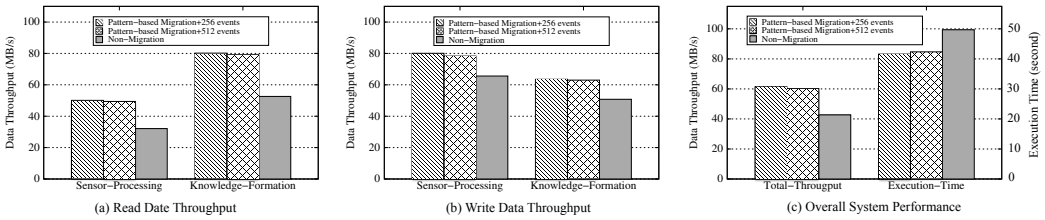
Fig. 6. The data throughput with varying number of client nodes, when the size of segment is 64KB. (a), (b), (c): Read data throughput in the different rounds of execution; (d), (e), (f): Write data throughput in the different rounds of execution.

strategy from achieving the same performance as the *Optimal Placement* strategy. On the other hand, when the access pattern has changed in the second round and the third round of execution, the *Optimal Static Placement* scheme works like *Non-Migration*, and the processes have to use the global file system for exchanging their data. For these rounds, the *Pattern-based Migration* strategy is able to dynamically adapt the placement of processes and it achieves the same performance as for the first round. The *Pattern-based Migration* thus outperforms the other strategies by up to 76 %.

Another interesting clue revealed in Figures 5 and 6, is that *Pattern-based Migration+256* slightly outperforms *Pattern-based Migration+512*. This is because the smaller time window indicates less time for disclosing a matched process pair, so that the process migration can be conducted a little earlier in our selected benchmark. However, it is possible to perform incorrect migrations when the time window is too small, even though this kind of case did not occur in our experiments.

2) Results of SSCA #3: In this series of experiments, the benchmark was configured with a scale factor of 6 and a dialed grid size of $4 \times 4 \times 4 = 64$ images on a single computing node, using the shared file system for storing the data. The size of the formed image is 1492×2286 Bytes, which means every node has to process 208MB of data, and all the 16 client nodes run the same jobs. We did not analyze the source codes of *SSCA #3*, so that the *Optimal Placement* scheme has not been used in this experiment.

Since the *SSCA #3* benchmark does not change its access



(a) Read Date Throughput (b) Write Data Throughput (c) Overall System Performance

Fig. 7. Results of running the *HPCS SSCA* #3 benchmark. (a)Read data bandwidth in different steps; (b)Write data bandwidth in different steps; (c)Overall system performance in the measurements of total data throughput and the execution time.

pattern in the different rounds of execution, the average statistics over the three rounds of execution are reported. Figures 7(a) and (b) reports the read data throughput, and write data throughput that were measured during the execution. Figure 7(c) presents the system performance by the measurements of the overall data throughput and the total execution time. From the reported results, we conclude that the proposed scheme of block access pattern-based process migration can effectively enhance the I/O data throughput by 42.7% compared to *Non-migration*, and reduce the application run time time by 14.8%. Besides, note that the two selected sizes of the time window adopted by the proposed mechanism do not make any significant performance difference in this experiment.

3) Case study of SPEEDY-LETKF: Because the first cycle of the *SPEEDY-LETKF* is the initialization cycle, and the access pattern does not change during the different cycles of execution, we run it with 2 cycles in each test, and we only report the results of the second cycle. In the tests, we set each *LETKF* ensemble member has 16 parallel processes. That indicates the number of SPEEDY processes is the number of *LETKF* ensemble members multiplied by 16. The *SPEEDY* processes are initially allocated from client node 0 to 15, following the round-robin fashion, and the *LETKF* processes are placed from client node 15 to node 0, following the round-robin fashion, as well. The *SPEEDY* model executes 1-month integration, and outputs the simulation results after the computation. The size of input data for the *SPEEDY* model is around 4.1MB, and the size of its output data, which is then read by the *LETKF* model, is 12.6MB. Furthermore, in order to explore the impact of different similarity thresholds and the value of *N* in the compressed matrix, on the system performance, we have also executed the benchmark with diverse settings of these parameters. As *Pattern-based Migration+256* and *Pattern-based Migration+512* have yielded the similar results in our experiments, only *Pattern-based Migration+256* has been used in this case study, and labeled as *Pattern-based Migration* by default. In this case study, we first measured the time required for running the application; next, we analyze the breakdown time; At last, the results in terms of I/O data throughput are presented.

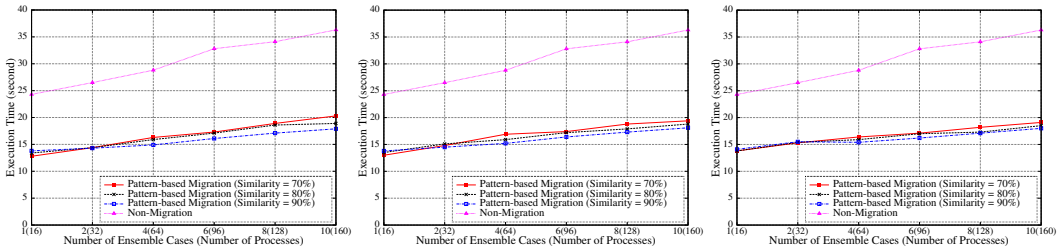
Time required for running the application. We measured the time required for executing the *SPEEDY-LETKF* application using the *Pattern-based Migration* and *Non-Migration* strategies in the storage system. In the experiments, we scaled the number of ensemble LETKF from 1 to 10, which indicates there are totally 16 to 160 processes for each kind of model. Figure 8 reports the time consumed for running the application benchmark utilizing the aforementioned two schemes. Especially, we employed the different scales of similarity and the value of *N* in the *Compressed Access Counting* scheme. As the figure shows, the proposed mechanism can significantly reduce the time needed for running the application, compared to the *Non-Migration* mechanism.

In order to analyze more precisely the execution time of *SPEEDY-LETKF*, when using *Pattern-based Migration*, we have conducted a breakdown analysis of the execution time. Figure 9 illustrates the results of time distribution. We conclude that the shorter execution time obtained when using *Pattern-based Migration*, is due to the I/O operations that are performed faster since the time required by computation remains the same. For example, when the size of ensemble instances is 10 (160 processes in total) in the test case, *Pattern-based Migration* only needs around a quarter of the I/O time consumed by *Non-Migration* to perform the pattern-based migrations, as well as the requested I/O operations.

Another interesting clue showed in Figures 8 and 9 is the impact of the *similarity threshold*, and the different value of *N* in the *Compressed Access Counting* scheme. Although varying *N* does not impact the performance significantly in the evaluation, the different scales of *similarity threshold* indeed play a role in performance fluctuations. When there are few processes, lower *similarity thresholds* achieve the best performance, whereas the performance obtained with higher *similarity thresholds* is improved as the number of processes grows. This is because the lower scale of *similarity threshold* requires less time for finding a matched pattern, but has a higher probability to migrate the process to the node that is not be the best choice. So that it may work well when there are few processes, but it does not cause attractive results when there are a large number of processes.

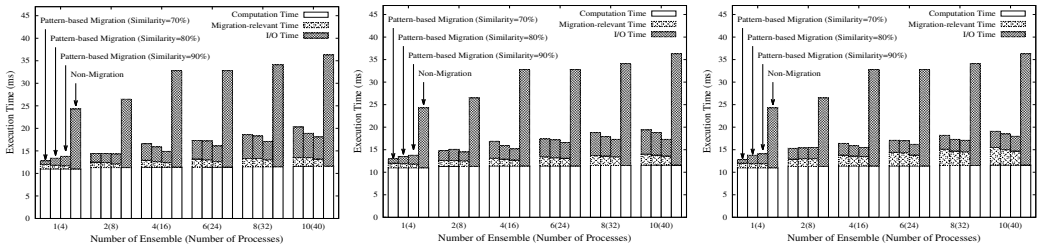
I/O Data Throughput. We have also measured the I/O data throughput, when running the test case. Figure 10 shows the data throughput of the two selected comparison strategies. The results show that the *Pattern-based Migration* strategy improves the I/O data throughput by up to 236.6 % compared to the *Non-Migration* scheme.

In summary, *SPEEDY-LETKF* communicates between processes in a pair-wise fashion, that indicates that the data written



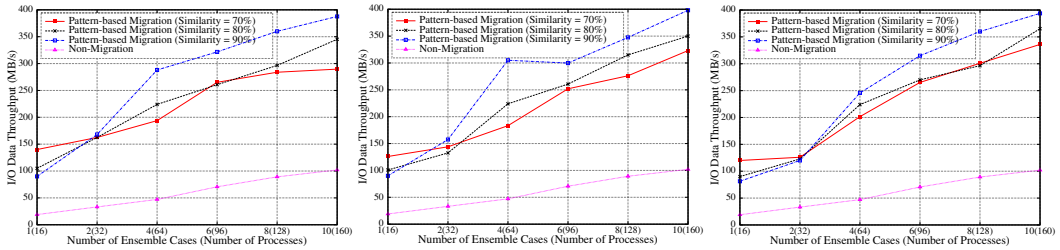
(a) Execution Time (N = 2) (b) Execution Time (N = 4) (c) Execution Time (N = 6)

Fig. 8. The total execution time when utilizing *Pattern-based Migration* and *Non-Migration*.



(a) Breakdown Time Analysis (N = 2) (b) Breakdown Time Analysis (N = 4) (c) Breakdown Time Analysis (N = 6)

Fig. 9. The breakdown time analysis when utilizing *Pattern-based Migration* and *Non-Migration*.



(a) I/O Data Throughput (N = 2) (b) I/O Data Throughput (N = 4) (c) I/O Data Throughput (N = 6)

Fig. 10. I/O data throughput when utilizing *Pattern-based Migration* and *Non-Migration*.

by a *SPEEDY* process is then read by the corresponding *LETKF* process. Consequently, the time required for exchanging the data between two kinds of processes can be remarkably reduced and the data rate can be significantly improved, when both of them are allocated in the same computing node, and the local file cache on the node is used for data exchange.

4) Overhead Analysis: The evaluation has shown that the pattern-based dynamical process migration can effectively and practically improve the performance for different workloads, but it is also necessary to measure the overhead caused the migration scheme we propose. In fact, in addition to conducing the process migration, the proposed mechanism also needs to analyze the block access events to profile the patterns, and computing the pattern similarity for triggering migration. This section evaluates the overhead caused by the migration of the processes between computing nodes, as well as the overhead.

Overhead on process migration.

Table II reports the overhead caused by the migration of processes, when running the benchmarks with default settings. Namely, the value of $Time_{window}$ is 256 block access events, and the value of *Similarity* is 0.9. Because the processes in the selected benchmarks have heavy data dependency, nearly half the processes have been migrated to other nodes. However, the results show that the overhead of the migration overhead is not high. For instance, *SPEEDY-LETKF* runnig 160 processes in total consumed 2.7 seconds of downtime for process migrations, but the migration saved more than 18 seconds of I/O time.

TABLE II
PROCESS MIGRATION OVERHEAD

Benchmark	# of Migrations	Traffic (MB)	Time (ms)
mpi_io_test	31	1400	1298
SSCA #3	7	742	1002
SPEEDY-LETKF (16)	8	422	1211
SPEEDY-LETKF (32)	16	825	1473
SPEEDY-LETKF (64)	32	1613	1794
SPEEDY-LETKF (96)	49	2211	2172
SPEEDY-LETKF (128)	65	2764	2445
SPEEDY-LETKF (160)	81	3187	2667

TABLE III
COMPUTATION AND SPACE OVERHEAD

Benchmark	**Time** (%)	**Space** (MB)
mpi_io_test	2.18	169.64
SSCA #3	1.25	26.98
SPEEDY-LETKF (16)	1.87	10.21
SPEEDY-LETKF (32)	1.81	22.18
SPEEDY-LETKF (64)	1.78	30.45
SPEEDY-LETKF (96)	1.76	42.16
SPEEDY-LETKF (128)	1.78	60.22
SPEEDY-LETKF (160)	1.77	77.44

While traditional checkpoint/restart mechanisms rely on disk operations, the *Pattern-based migration* transfers the state of a process to the destination client node using the MPI communication facility. As a result, the process can be resumed to execute on the destination node with a short time interruption. In other words, the downtime of the migrated process is reduced when using MPI for transfering the process state. This downtime is compensated with the I/O operations that are carried out faster, we shown in the previous sections.

Overhead on computation and storage. The *Pattern-based Migration* mechanism records the block access events and then profiles block access patterns using the *Access Counting Scheme*, in order to calculate the pattern similarity and to direct the process migration. Table III reports the overhead on computation and space storage due to the *Access Counting Scheme* for the benchmarks presented in the previous sections.

The results show that the overhead in terms of computation is minor. The overhead in terms of space storage remains low: less than 200 MB of disk space is used for storing trace logs. Thus, analyzing the similarity of block access patterns and activating the process migration can be done on the same machine as the storage system without causing too much computation and disk overhead. For long-time running applications, the size of trace logs may become extraordinary large, in this case, the *Pattern-based Migration* scheme may discard certain logs that occurred early, because disk block correlations are relatively stable during certain period, and the disk access logs at earlier stages do not reflect the current access patterns [Li 2004].

V. CONCLUSION

This paper has proposed, implemented, and evaluated a novel process migration mechanism for yielding better I/O performance. It leverages the block access pattern similarity of processes, which is computed by the storage servers, to properly perform migrations. The experiments have shown that this mechanism can benefit to coupled applications or other scientific applications running on high performance clusters, without any modifications to the application themselves. To put it from another angle, we first profile the block access pattern of processes for a specified period using the scheme of *Access Counting Diagram*. Next, we calculate the pattern similarity of the patterns of process pairs and search for processes that have similar access patterns. When such a pair of processes is found, one of the processes is migrated to the other process computing node. Therefore, both processes can exchange data via the local file cache on the computing node, instead of using the global file system. The results of the evaluation experiments show that not only the I/O data throughput can be remarkably improved, but also the execution time of the application can be greatly reduced.

We also emphasize that the idea of process migration according to pattern similarity presented in this paper can be applied to other conventional distributed/parallel file systems such as Lustre, the Google file system, PVFS or the Hadoop distributed file system. Our current implementation selects the targeted migration process on the basis of I/O workloads, and the computing workloads is not considered. This may result in migrating an I/O-bound process running on an idle computing node, to a busy one, though this kind of cases did not appear in the evaluation experiments. Moreover, the intra-application communications among processes are not taken into account before conducting process migrations. Therefore, we are intending to optimize the implementation, and using the information about the computing utilization on the computing nodes and the intra-application communication traffics, to properly carry out process migrations. Besides, intelligently adapting the values of parameters used in the proposed mechanism for the different application contexts, is another direction of our future work.

VI. ACKNOWLEDGEMENTS

This work was partially supported by "National Natural Science Foundation of China (No. 61303038)", "Natural Science Foundation Project of CQ CSTC (No. CSTC2013JCYJA40050)", "the Scientific Research Foundation for the Returned Overseas Chinese Scholars, State Education Ministry", and "the Opening Project of State Key Laboratory for Novel Software Technology (No. KFKT2014B17)".

REFERENCES

[1] Bailey D., Barszcz E., and Barton J. et al. The NAS parallel benchmarks. International Journal of High Performance Computing Applications, Vol. 5(3): 63–73, 1991.

[2] Chen H., Chen W., and Huang J. et al. MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multi-clusters. In Proceedings of the 20th annual international conference on Supercomputing (ICS '2006), pp. 353–360, 2006.

[3] Chen Q, Guo M. Adaptive workload-aware task scheduling for single-ISA asymmetric multicore architec- tures. ACM Transactions on Architecture and Code Optimization (TACO), Vol. 11(1): 8, 2014.

[4] Cheng L., Wang C. vBalance: using interrupt load balance to improve I/O performance for SMP virtual machines. In Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '2012), ACM, 2012.

[5] Clark C., Fraser K., and Hand S., et al. Live migration of virtual machines. In Proceedings of the 2nd conference on Symposium on Networked Systems Design and Implementation (NSDI '2005), USENIX Association, pp. 273-286, 2005.

[6] Costa B., Al-Kiswany S., and Yang H., et al. Supporting storage configuration for i/o intensive workflows. In Proceedings of the 28th ACM international conference on Supercomputing (ICS '2014). ACM, pp. 191–200, 2014.

[7] Dean J. and Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. Communication of ACM, Vol. 51(1):107–113, 2008.

[8] Ding X., Jiang S., Chen F., and Zhang X. et al. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In Proceedings of USENIX Annual Technical Conference (ATC '2007), San Francisco: USENIX Association, 2007.

[9] Dong X., Xie Y., and Muralimanohar N. et al. Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. ACM Transactions on Architecture and Code Optimization (TACO), Vol.8(2): 6, 2011.

[10] Dongarra J. and Beckman P. et al. The International Exascale Software Roadmap. International Journal of High Performance Computer Applications, Vol. 25(1), pp. 3–60, 2011.

[11] Du C., Sun X., and Wu M. Dynamic scheduling with process migration. Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGRID '2007), pp. 92–99, 2007.

[12] Filesystem in Userspace. http://fuse.sourceforge.net/ [Accessed in Nov., 2010]

[13] Gong Z., Boyuka D., and Zou X. et al. Parlo: Parallel run-time layout optimization for scientific data explorations with heterogeneous access patterns. In Proceedings of 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '2013), IEEE, pp. 343–351, 2013.

[14] Hack J., Caron J., and Danabasoglu G. et al. CCSM-CAM3 climate simulation sensitivity to changes in horizontal resolution. Journal of climate, Vol. 19(11): 2267–2289, 2006.

[15] He J., Bent J., Torres A., and Sun X., et al. I/O acceleration with pattern detection. In Proceedings of the 22nd international symposium on High-performance parallel and distributed computing (HPDC '2013). ACM, New York, NY, USA, pp. 25–36, 2013.

[16] HPCS challenge benchmarks scalable synthetic compact application. SSCA #3: Sensor processing and knowledge formation, MIT. http://www.highproductivity.org/SSCABmks.htm, 2005.

[17] Hunt B., Kostelich E., and Szunyogh I. Efficient data assimilation for spatiotemporal chaos: A local ensemble transform Kalman filter. Physica D: Nonlinear Phenomena, 2007, Vol.230(1): 112–126.

[18] Jeannot E., Mercier G., and Tessier F. Process Placement in Multicore Clusters:Algorithmic Issues and Practical Techniques. IEEE Transactions on Parallel and Distributed Systems, Vol. 25(4):993–1002, 2014.

[19] Jiang S., Ding X., Xu Y., and Davis K. A Prefetching Scheme Exploiting both Data Layout and Access History on Disk. ACM Transaction on Storage, Vol. 9:Article 10, 2013.

[20] Larson J., Jacob R., and Ong E. The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models. International Journal of High Performance Computing Applications, Vol.19(3):277–292, 2005.

[21] Li Z., Chen Z., Srinivasan S., and Zhou Y. C-Miner: Mining Block Correlations in Storage Systems. In Proceedings of the Third USENIX Conference on File and Storage Technologies (ATC '2004), San Francisco: USENIX, 2004.

[22] Liao J. A New Concurrent Checkpoint Mechanism for Embeded Multi-Core Systems. Computing and Informatics, Vol. 31(3): 693-709, 2012(a).

[23] Liao J., and Ishikawa Y. Partial Replication of Metadata to Achieve High Metadata Availability in Parallel File Systems. In the Proceedings of 41st International Conference on Parallel Processing (ICPP '2012), pp. 168-177, 2012(b).

[24] Liao J., Trahay F., Gerofi B., and Ishikawa Y. Prefetching on Storage Servers through Mining Access Patterns on Blocks. IEEE Transactions on Parallel and Distributed Systems, DOI: 10.1109/TPDS.2015.2496595, 2015.

[25] Meneses E., Ni X., Zheng G., and Mendes C. et al. Using migratable objects to enhance fault tolerance schemes in supercomputers. IEEE Transactions on Parallel and Distributed Systems, Vol. 26(7):2061–2074, 2015.

[26] Mercier, G., Clet-Ortega, J. Towards an efficient process placement policy for MPI applications in multicore environments. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) PVM/MPI. LNCS, vol. 5759, pp. 104–115. Springer, Heidelberg, 2009.

[27] Mercier G. and Jeannot E. Improving MPI applications performance on multicore clusters with rank re-ordering. Recent Advances in the Message Passing Interface, pp. 39-49, 2011.

[28] Michaud P., Seznec A., Fetis D., Sazeides Y., and Constantinou T. A study of thread migration in temperature-constrained multicores. ACM Trans. Archit. Code Optim. Vol.4(2), Article 9, 2007.

[29] Molteni F. Atmospheric simulations using a GCM with simplified physical parametrizations. I: Model climatology and variability in multi-decadal experiments. Climate Dynamics, Vol. 20, pp. 175-191, 2003.

[30] MPI-IO Test (fs test). http://institute.lanl.gov/data/software/ [Accessed in May, 2014]

[31] MPICH2, https://www.mpich.org/ [Accessed in Dec., 2012]

[32] Piernas J., and Nieplocha J.. Implementation and evaluation of active storage in modern parallel file systems. Parallel Computing, Vol.36(1): 26-47, 2010.

[33] Piernas J., Nieplocha J., and Felix E. Evaluation of active storage strategies for the lustre parallel file system. In Proceedings of the 2007 ACM/IEEE conference on Supercomputing (SC '2007), 2007.

[34] Pilla L., Ribeiro C., and Cordeiro D. et al. A hierarchical approach for load balancing on parallel multi-core systems. In Proceedings of the 41st International Conference on Parallel Processing (ICPP 2012), pp. 118-127, 2012.

[35] Valcke S., Budich R., and Carter M. et al. The PRISM software framework and the OASIS coupler. In proceedings of the 18 Annual BMRC Modelling Workshop, Melbourne, Australia, 2006.

[36] Valcke S., Balaji V. Craig A., and Riley G. et al. Coupling Technologies for Earth System Modelling. Geoscientific Model Development, 2012.

[37] Vecchiola C., Pandey S., and Buyya R. High-performance cloud computing: A view of scientific applications In Proceedings of the 10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN '2009), pp. 4-16, 2009.

[38] Wang C., Mueller F., and Engelmann C., et al. Proactive process-level live migration in HPC environments. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '2008). IEEE Press, 2008.

[39] Williams D., Jamjoom H., and Weatherspoon H. The Xen-Blanket: virtualize once, run everywhere. In Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '2012). ACM, New York, NY, pp. 113–126, 2012.

[40] Xie Y., Feng D., Li Y. and Long D. Oasis: An active storage framework for object storage platform. Future Generation Computer Systems, 2015.

[41] Zheng F., Zou H., and Eisenhauer G. et al. Flexio: I/O middleware for location-flexible scientific data analytics. In Proceedings of IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS '2013),pp. 320-331, 2013.