File Clustering Based Replication Algorithm in a Grid Environment

Hitoshi Sato[†], Satoshi Matsuoka[†,‡], and Toshio Endo[†]
[†] Tokyo Institute of Technology
[‡] National Institute of Informatics
{hitoshi.sato, matsu, endo}@is.titech.ac.jp

Abstract

Replication in grid file systems can significantly improves I/O performance of data-intensive applications. However, most of existing replication techniques apply to individual files, which may introduce inefficient replication overheads for a large number of files. We propose a file clustering based replication algorithm for grid file systems. Our algorithm groups files according to a relationship of simultaneous accesses between files and stores the replicas of the clustered files into storage nodes, to satisfy expected most of future read access times to the clustered files and replication times for individual files being minimized under the given storage capacity limitation. Our experiments on a given grid environment, 20 nodes of 5 sites, suggest that the proposed algorithm achieves accurate file clustering and efficient replica management; our clustering policy with the file cluster size limit of 5120 MB and storage capacity limit for replicas of 10240 MB exhibits 1.58 times efficiency than the policy that never cluster related files. The results also indicate that the overheads required for introducing our algorithm significantly affect I/O performance of running applications.

1. Introduction

Large-scale grid environments are becoming viable platforms for data-intensive applications, since they can provide much larger amounts of computational power and storage space than those of typical single-site environments. One of the difficulties in data sharing on these environments, however, is the complexity of dealing with multiple distributed data sources. Often different sites provide different file systems, so the users are forced to manually stage data in an ad-hoc, complicated fashion.

Grid file systems can solve this difficulty by providing a single-system image to application users. However, application performance can be substantially degraded due to such problems as access contentions and costly remote file accesses. Existing approaches to these problems, including replication and caching [12, 15, 14, 3], employ rather fixed and simple heuristics and may not yield optimal access performance in dynamically changing grid environments. Our earlier work [19] has presented an optimal replication algorithm by modeling the determining replication problem as integer linear programming (ILP) problems. Nevertheless, these existing approaches apply to individual files in a grid file system, which may introduce inefficient replication operations for a large number of files.

Clustering related files can achieve efficient replica management, since file operations such as replication, migration, and deletion, can be applied to the clustered file sets collectively. For example, Doraimani et al. [13] have shown that *scientific data usage translates into requests for groups of correlated files* from real application traces in high-energy physics [8]. The accuracy of file clustering is dependent on how to model the relationships between files. Most previous approaches [16, 10] focus *recency* or *frequency* based file clustering; however, existing file-location-aware job scheduling behavior [11, 20] and underlying recent many-core architecture computing nodes disturb accurate file clustering, since data-intensive job schedulers tend to allocate submitted jobs to individual cores on nodes where necessary files can be accessed locally, which may cause file access contention on a single node when popular but unrelated files are accessed simultaneously.

To facilitate efficient data management in grid file systems, we propose a data replication technique based on clustering related files in a grid file system. Our algorithm groups files according to a relationship of simultaneous accesses between files; we determine file clusters accessed at once by a single process to exclude the influences of the file accesses of other processes. We then store the replicas of the file clusters into storage nodes to satisfy expected most of future read access times to the clustered files and replication times for the individual files being minimized under the given storage capacity limitation. We model this replication problem as a combination of a graph partition problem and ILP problems, where constraints are derived from

the summation of file size in a file cluster and the allowable storage capacity for replicas, and objectives are to minimize expected read access times to the clusters and replication times for the individual files. We determine the replication strategy by solving these cost minimization problems, estimating inter-node link throughputs, access popularities to files, and access relationships between files. We obtain this information by monitoring run-time behavior of file accesses from applications.

We evaluate our algorithm on a live grid environment [5] composed of 5 sites. Comparison with other clustering techniques and replication policies shows that our algorithm succeeds in accurate file clustering and efficient replica management; our clustering policy with the file cluster size limit of 5120 MB and the storage capacity limit for replicas of 10240 MB exhibits 1.58 times efficiency than the policy that never cluster related files. The results also indicate that the overheads required for introducing our algorithm significantly affect I/O performance of running applications.

2. Related Work

One of the most effective techniques to improve file I/O performance in grid file systems is to deploy related files to closest network proximity of the requesting nodes. In particular, *read access performance* can significantly affect application turn-around times, since data-intensive applications often read large data but only write a small amount of results. Therefore, improving read access performance is especially effective for such write-once, read mostly applications. However, determining optimal replication, i.e., which files should be replicated to where, is a difficult problem, because they depend on underlying network performance characteristics and applications of file access patterns, which may only be observed at application run times.

Existing network file systems that support file replication mechanisms [12, 15, 14, 21, 3] do not always optimally distribute replicas, since they statically determine replica locations without considering both network topologies and application performance characteristics. We have proposed a replication mechanism that considers both application workloads and wide-area network performance [19]. However, these mechanisms apply to individual files (or file segments) stored in a network file system, which may introduce inefficient replication overheads for a large number of files.

File caching in grid file systems is similar to caching in standard file systems. For example, optimally selecting files to cache has to consider how likely they are used again. However, one of the difficulties that is unique to grid file systems is that optimal caching also depends on where cached files are originally located. Moreover, cache usage exhibits different behavior according to running applications. Therefore, optimal caching strategies need to consider both access patterns and network properties.

There have been a number of efforts in such data management techniques for grids [22]. Ranganathan et al. [17] presented a simulation analysis of replication techniques based on data-intensive workloads. Although dynamic replication helps to reduce hotspots created by popular files, actual file access performance can be inefficient, especially on heterogeneous large-scale environments. Rahman [18] et al. and Wang et al. [23] solve similar replication or migration problems as model-based combinational optimization problems. However, most of the existing approaches focus replication for individual files. Doraimani et al. have shown that real data-intensive application workloads exhibit relationships between files in their usage and proposed a caching algorithm that considers groups of files always used together in data processing jobs [13]. These works emphasis the effectiveness of file clustering based data management; however, the approach does not optimize overall file system performance.

Clustering related files is a traditional technique in network file systems [16, 10]. Existing proposals mainly employ *recency* [16] or *frequency* [10] as a metric to represent the strength of access relationships between files; *recency* based techniques consider the time series of file accesses, i.e., files that are accessed recently are regarded as a single file cluster, while *frequency* based techniques consider the order of file accesses, i.e. files that are accessed successively are regarded as a single file cluster. However, file clustering accuracy in these techniques is significantly influenced by application workloads, which may vary from system to system in different times and timescales. For example, existing data-intensive job scheduling systems tend to allocate jobs to individual cores on nodes where necessary files can be accessed locally for avoiding costly remote file accesses [11, 20]. However, such job scheduling can cause contention of compute and I/O resources when popular files are simultaneously accessed; many jobs can be scheduled on a small number of nodes at the same time.

3. File Clustering Based Replication Algorithm

Our proposed technique automatically determines optimal file replication strategies, which groups files according to a relationship of simultaneous accesses between files and stores the replicas of the clustered files into storage nodes, to satisfy expected most of future read access times to the clustered files and replication times for the individual files being minimized under the given storage capacity limitation. To do so, it passively monitors file access performance of running applications and estimates inter-node link throughputs, access popularities to files, and access relationships between files, from the observed performance

data.

3.1. Access Pattern Detection

Our strategy needs to collect the following file access information: timestamp, path, access type (*i.e.*, read or write), I/O size, elapsed time of each file access, destination hostname, and process identifier (PID) that accesses the file. We have developed a library-based tracer to capture the information on every file access [19]. The library function is called on every file operation and logs the profile into a performance database. We then use these profiles to estimate inter-node link throughputs, access popularities to files, and access relationships between files. We will explain implementation details in Section 4.

3.2. Throughput Estimation

We estimate network link throughputs by using collected access profiles, including I/O sizes and elapsed times. We use a model to estimate elapsed times described in [19]: $time = \frac{1}{thput} \times io\_size + e$, where $time$ corresponds to elapsed times, $thput$ to a link throughput between nodes, and io_size to I/O sizes. Real access times can be perturbed by various noises; we include e to accommodate such noises. For each link, we estimate its throughput by fitting this linear model to collected access profiles with the least square method. Finally, we create a throughput matrix, $D = (d_{i,j})$, where $d_{i,j} = \frac{1}{thput}$ represents the reciprocal of estimated throughput between nodes i and j.

3.3. Access Popularity Estimation

We introduce file access probability to estimate future file accesses. For simplicity, assuming that time intervals of file accesses from a client node to a file obey an exponential distribution with parameter λ, the probability density function of the distribution is described as follows:

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x} & (x \geq 0) \\ 0 & (x < 0) \end{cases}$$

Based on collected access profiles, including timestamp and access type (i.e., read or write), we can estimate time intervals of file accesses. We estimate the parameter λ by fitting this distribution model to the time intervals with the Bayesian or maximum likelihood inference. Let $\dot{\lambda}$ be the determined parameter, t_0 be the last time when a file is accessed from a client node, and t be the current time. The estimated future file access probability from node i to file k, $p_{i,k}$, is described as follows:

$$p_{i,k} = \int_{t-t_0}^{\infty} \dot{\lambda} e^{-\dot{\lambda}x} dx = e^{-\dot{\lambda}(t-t_0)}$$

For example, if node i accesses file k recently, $p_{i,k}$ is asymptotic to 1, while if node i is unlikely to access file k, $p_{i,k}$ is asymptotic to 0. We employ this parameter $p_{i,k}$ to estimate file access popularity for file k from node i.

3.4. Access Relationship Estimation

Based on collected access profiles, we group stored files according to a relationship of simultaneous file accesses. We assume *files that are accessed by a single process simultaneously tend to be accessed in the near future*. Therefore, we determine file clusters that a single process is likely to access simultaneously. Note that the file cluster has a limit of its total file size, because our objective in this phase is to divide files stored in a file system into several file clusters for data management, where storage nodes have capacity constraints.

To represent the relationship between files, we count the number of times that a single process running on a client node accesses to any two files, using collected access profiles: path and PID. Then, we create a weighted graph where a weighted vertex represents a file and its size and a weighted edge represents the number of times of simultaneous file accesses by a single process. We solve this file clustering problem as a graph partition problem, where the objective is to divide the graph into several sub-graphs such that the summation of vertex weights in the sub-graph, which we denote as *cluster weight*, is less than specified threshold and the summation of edge weights between the sub-graphs, which we denote as *cut weight*, is minimized.

We determine file clusters with the file access relationship as follows. First, we divide the graph into several sub-graphs such that the cut weight equals zero. Since such file clusters are regarded as no relationship in terms of simultaneous file accesses, we employ each of these file clusters for an optimal unit for data management, i.e., replication, migration and deletion, respectively. Then, we divide each divided sub-graph (graph) into several sub-graphs such that the cluster weight is less than the specified threshold, by dividing the target graph into two sub-graphs iteratively until the sub-graphs satisfy the cluster weight.

By solving the above problem, we determine file clusters in which files are likely to be accessed simultaneously.

3.5. Determining Replica Locations

Using the estimated inter-node link throughputs, file access popularities, and access relationships between files, we determine replica locations of the file clusters, which satisfies expected most of future read access times to the file clusters to be minimized under the given storage capacity limitation. We model this replica location determination problem as an ILP problem. Here, we assume that many

data-intensive applications read through the entire contents of accessed files for simplicity.

We determine the replica locations of file cluster F as follows. Let $i \in \{1, \ldots, n\}$ be the node names in the target environment, and the capacity of node i be c_i. Let $k \in \{1, \ldots, m\}$ be the file names in file cluster F, and the size of file k be s_k. We denote the availability of file cluster F on node i as x_i, where $x_i \in \{0, 1\}$. x_i being 1 denotes that node i has a replica of file cluster F, while x_i being 0 denotes that node i does not.

We model the constraint on the number of replicas for file cluster F as follows. Let R be the given storage capacity limitation for file cluster F, and s be the summation of file size in file cluster F, i.e., : $s = \sum_{k=1}^{m} s_k$, which denotes cluster weight of file cluster F. Thus the constraint on the number of replicas for file cluster F can be denoted as:

$$1 \leq \sum_{i=1}^{m} x_i \leq \frac{R}{s} + 1$$

Note that we guarantee the availability of file cluster F; at least one replica of file cluster F exists in the file system. Then we model file read access times in file cluster F as follows. We denote accesses to file cluster F from node i to node j as $y_{i,j}$, where $j \in \{1, \cdots, n\}$.

Then we model file read access times as follows. We denote accesses to file k in file cluster F from node i to node j as $y_{i,j,k}$, where $j \in \{1, \cdots, n\}$. $y_{i,j,k}$ being 1 denotes that node i accesses file k on node j, and $y_{i,j,k}$ being 0 denotes that node i does not. Here, we assume each node only accesses a single node that holds a replica of file k. Then, estimated access times that can be taken to read file k of size s_k available on node j from node i can be denoted as $s_k \cdot d_{i,j} \cdot y_{i,j,k}$ Recall that $d_{i,j}$ denotes the reciprocal of the estimated throughput between nodes i and j. For example, if node i accesses file k in its local storage or does not access file k on node j, the estimated access time is equal to 0, since $d_{i,j} = 0$ or $y_{i,j,k} = 0$. On the other hand, if node i actually accesses file k on node j, the time is $s_k \cdot d_{k,j}$, where we assume that read access times can be estimated by dividing the file size by the link throughput. We further extend this model by considering estimated access popularities. Let $p_{i,k}$ be the estimated future file access probability from node i to file k. The expected mean read time to file k from node i to node j, $t_{i,j,k}$, can be denoted as:

$$t_{i,j,k} = p_{i,k} \cdot s_k \cdot d_{i,j} \cdot y_{i,j,k}$$

For example, if node i is unlikely to access file k, $t_{i,j,k}$ is relaxed, since $p_{i,k}$ is asymptotic to 0. Thus the expected mean read time to file cluster F from node i to node j, $t_{i,j}$, can be denoted as:

$$t_{i,j} = \sum_{k=1}^{m} t_{i,j,k}$$

Our objective in this phase is to minimize expected most of future read access times to files in file cluster F. Let t be the maximum read access time to cluster F from node i to node j, i.e.:

$$t = \max_{i,j \in \{1, \cdots, n\}} t_{i,j}$$

We minimize t under the given storage capacity limitation R and the constraints as stated above.

Tying the objective function and the constraints together, we can model them as the following ILP problem:

Minimize

$$t = \max_{i,j \in \{1, \cdots, n\}} t_{i,j} \tag{1}$$

Subject to

$$x_i \in \{0, 1\} \tag{2}$$

$$y_{i,j,k} \in \{0, 1\} \tag{3}$$

$$1 \leq \sum_{i=1}^{n} x_i \leq \frac{R}{s} + 1 \tag{4}$$

$$s \cdot x_i \leq c_i \tag{5}$$

$$s = \sum_{k=1}^{m} s_k \tag{6}$$

$$t_{i,j} = \sum_{k=1}^{m} t_{i,j,k} = \sum_{k=1}^{m} p_{i,k} \cdot s_k \cdot d_{i,j} \cdot y_{i,j,k} \tag{7}$$

$$\sum_{j=1}^{n} y_{i,j,k} = 1 \tag{8}$$

$$x_j = 1, \text{if} \sum_{i=1}^{n} y_{i,j,k} > 0 \tag{9}$$

The objective function (1) minimizes expected most of future read access times to the clustered files. In addtion to the previously described constraints: (2), (3), (4), (6), and (7), we include three other constraints in this problem. First, constraint (5) states that the cluster size must not exceed the capacity of local storage, if the node holds a replica of the file cluster. Second, constraint (8) states that each node only accesses a single node that holds a replica of file k. Finally, constraint (9) states that if there are accesses from any nodes to a node, the accessed node must hold a file cluster.

By solvinng the above problem, we make decision on the nodes that hold replicas of file cluster F by determining the values of x_i for all $i \in \{1, \cdots, n\}$.

3.6. Determining Replica Movement

Now that we have identified which nodes will have replicas of file cluster F that includes file k of size s_k for $k \in \{1, \cdots, m\}$, we determine how to efficiently create these new replicas using current replica placements. In other words, we minimize the time to transfer file replicas in file cluster F over distributed nodes. Let l_i be the new locations for file cluster F, which is an alias of the determined value, x_i, and $\bar{l_{i,k}}$ be the current locations for file k. The time, q_i, that node i will take for transferring file replicas can be represented as:

$$q_i = \sum_{j=1}^{n} \sum_{k=1}^{m} s_k \cdot d_{i,j} \cdot z_{i,j,k}$$

, where $z_{i,j,k} \in \{0, 1\}$ denotes whether node i transfers a replica of file k to node j, i.e., if $z_{i,j,k} = 1$, node i does transfer a replica of file k to node j, and if $z_{i,j,k} = 0$, otherwise. In addition, let q be the maximum transfer time of each node, i.e.,:

$$q = \max_{i=1,\cdots,n} q_i$$

Our objective in this phase is to minimize q, since we would like to minimize the overall replica transfer time. In summary, the replica movement problem can again be modeled as the following ILP problem:

Minimize

$$q = \max_{i=1,\cdots,n} q_i \tag{10}$$

Subject to

$$z_{i,j,k} \in \{0, 1\} \tag{11}$$

$$q_i = \sum_{i=1}^{n} \sum_{k=1}^{m} s_k \cdot d_{i,j} \cdot z_{i,j,k} \tag{12}$$

$$\sum_{j=1}^{n} z_{i,j,k} \geq 0, \text{if } \bar{l_{i,k}} = 1 \tag{13}$$

$$\sum_{i=1}^{n} z_{i,j,k} = l_j \tag{14}$$

The objective function (10) minimizes the maximum transfer time between file system nodes. Here, we also include two additional constraints with previously described constraints (11) and (12). Constraint (13) states that if there exists a transfer for a replica of file k from node i to any nodes, node i must already have the replica, and constraint (14) states that the existence of a new replica on node j depends on whether any node transfers file k to node j.

By solving the above problem, we determine the values of $z_{i,j,k}$ for all $i, j \in \{1, \cdots, n\}, k \in \{1, \cdots, m\}$. Our system transfers replicas of files in file cluster F according to these results.

4. Implementation

We implemented the proposed algorithm in our previous prototype file system [19]. We also improve the previous prototype architecture to be able to support various grid file systems [1, 9, 3]. This section describes the detail of our implementation for the proposed algorithm.

We implemented a library-based tracer to capture information on every file access in a file system. Most grid file systems provide a library to mount the file system via FUSE mechanism, which is a framework for implementing and extending file system interfaces in user space programs. In the FUSE-based file systems, each system call to standard file system interfaces, such as open and close, is redirected to FUSE handlers. We use this mechanism to encapsulate differences in implementation of monitoring mechanisms between existing grid file systems.

When an application on a client first opens a file, we allocate an entry for the file access in the profile database and assign a key for the entry. Here, we record the timestamp, path, and PID of file opening in the entry. Note that this operation is only done when files are first opened (or created). The PID of the file access is collected from a function `fuse_get_context` provided by FUSE. For each future read and write access, we look up the entry for the file access by using the unique structure as its key and record the timestamp into the entry. When the application last closes the file, we record the destination hostname of the file access. Our library efficiently implements this mechanism by delegating all database operations to concurrent threads, which in turn operates with a SQLite based database.

We group files stored in the file system and determine which file clusters to replicate to where using the algorithm described in Section 3. The introspection mechanism is implemented as a daemon process running on a node. The daemon periodically queries the meta-data server and the profile databases on client nodes to obtain meta-data of files and collected access profiles. Based on the information, we apply our algorithm to estimate the file system state: inter-node throughputs, access popularities to files, and access relationships between files. Our prototype currently uses METIS [6] for solving graph partition problems and GNU GLPK [2] for solving ILP problems.

The determined results are applied by a introspection daemon, which implements file operations, including replication and deletion, by using commands or libraries of the underlying grid file system. We encapsulate differences in implementation of file operations between grid file systems

Table 1. Specification of Cluster Nodes.

Site	hongo, chiba	keio	tohoku, tsukuba
CPU type	Core 2 Duo	Xeon E5410	
Cycle	2.13 GHz	2.33 GHz	
#Cores	2	8	
Memory	4GB	16GB	32GB
OS	Linux 2.6.18		
Network	Gigabit Ethernet		

in this layer.

We applied this monitoring and introspection mechanisms to the Gfarm file system (Gfarm), which is a client-server-based grid file system. Gfarm provides a program to mount a Gfarm file system using FUSE library. We modify this mount program by using our trace library. We also include file operations for introspection to our framework by using Gfarm commands and libraries, such as `gfrep` and `gfrm`.

5. Experiments

To evaluate the effectiveness of our proposed replication technique, we apply Gfarm-based prototype to the InTrigger grid environment [5], which is a collection of clusters distributed across Japanese universities and national laboratories. We deploy our prototype file system over 5 clusters located in distributed sites: hongo (Tokyo), chiba (Chiba Pref.), keio (Tokyo), tohoku (Miyagi Pref.), and tsukuba (Ibaraki Pref.). The specifications of each cluster node are shown in Table 1, and the network performances between sites are shown in Table2. We designate a single node in the chiba site as a meta-data server and use the remaining nodes, 4 nodes in each site, as both clients and file-system nodes. We conducted data-intensive workloads on the prototype, which perform burstry accesses to files in a data set. Figure 1, where the x-axis corresponds to data set IDs, while the y-axis to file size distribution in the data set; we set six data sets (9761 MB in total). This behavior is collected from BLAST [7], which is a similarity search application for nucleotide or protein sequence databases; this kind of workloads is commonly seen in many data-intensive applications. Each job reads files in a data set, which are initially located on a single node in the chiba site. Note that the experiments are conducted under the non-dedicated live grid environment, which may be affected other workloads.

First, we compare our file clustering techniques, described in Section 3.4, with two alternative techniques:

- Files that are recently accessed within n (= 1) sec in a single node belong to the same file cluster: (*recency*-n)

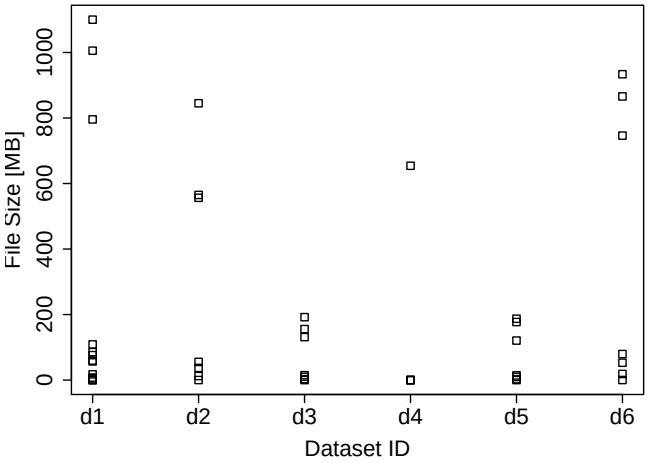

Figure 1. File Size Distributions.

- Files that are accessed successively more than n (= 5) $times$ in a single node belong to the same file cluster: (*frequency*-n)

Our proposal limits the total file size in a file cluster to n (= 2048) MB: (*pid*-n). We conduct BLAST workloads on a single node in the keio site, which access each data set 10 $times$ during 1800 sec.

Figure 2 shows the results of the file clustering. The x-axis corresponds to the total file size in a file cluster, and the y-axis corresponds to the degree of accuracy in the clustered files, which represents how different files of data sets are included in a single file cluster; we denote this value as the reciprocal value of the number of data sets included in a single file cluster, i.e., if the value being 1 denotes that the file cluster consists of a single data set, while the value being $\frac{1}{n}$ denotes the file cluster consists of n data sets. We observe that the *recency*-1 and *frequency*-5 techniques generate inaccurate file clusters: 7151 MB with 0.17 in the *recency*-1 technique and 7058 MB with 0.2 in the *frequency*-5 technique. On the other hands, our proposal (*pid-2048*) exhibits a good clustering that consists of a single data set under the given file size limitation.

Next, we conduct different BLAST workloads that consist of two time interval phases (1 $phase$ = 900 sec) as followings:

Phase1 Nodes in the hongo and chiba sites access d1, d2, and d3 data sets, while nodes in the keio, tohoku, and tsukuba sites access d4, d5, and d6 data sets

Phase2 Nodes in the hongo and chiba sites access d4, d5, and d6 datasets, while nodes in the keio, tohoku, and tsukuba sites access d1, d2, and d3 data sets

Table 2. Network Performance. [RTT (ms) / Bandwidth(MB/s)]

src / dst	hongo	chiba	keio	tohoku	tsukuba
hongo	0.20 / 112	6.28 / 104	6.83 / 7.03	14.4 / 85.1	7.69 / 36.6
chiba	6.39 / 71.1	0.190 / 112	13.0 / 7.60	18.7 / 70.9	11.0 / 73.5
keio	7.43 / 2.14	12.8 / 0.75	0.0510 / 99.8	21.1 / 0.36	13.3 / 0.95
tohoku	14.5 / 2.58	18.8 / 1.37	20.8 / 3.30	0.170 / 104	9.41 / 34.3
tsukuba	7.78 / 4.72	11.2 / 7.91	13.1 / 6.92	9.49 / 8.85	0.22 / 110

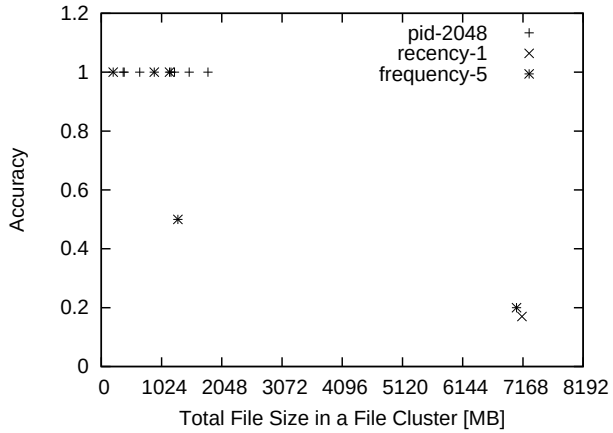

Figure 2. File Clustering.

Each node submits jobs that access target data sets 5 times during each time phase. We compare our proposal, the limit of a file cluster is n (= 1024, 5120) MB: (*pid-*n), with following alternative policies:

- No replication is performed (*no_rep*). Each node accesses a single remote node that holds the target files.
- No clustering is performed (*no_clustering*). The policy applies our algorithm described in Section 3 without any clustering techniques.

We set the storage capacity limit for replicas to 10240 MB. Our algorithm shares the allowable storage capacity according to the total size of the file clusters. We also set the time interval to apply each data management policy to 300 sec.

Table 3 shows the performances of each data management policy, including the average throughput of remote file accesses (io thput), the maximum usage of storage (st size), the average throughput of replica transfer (rep thput) in which the total transfer size is divided by the total transfer time, the total time of determining replica locations (dl time), and the total time of determining replica movement (dm time). We see that both the *clustering* and *no_clustering* policies automatically determine appropriate nodes under the given storage capacity limitation. We also observe that there are no significance differences in the remote I/O throughput between the *clustering* and *no_clustering* policies. This is due to the overheads of determining replication, including both locations and movement. Delayed determination introduces ineffective data management, since required replicas are not supplied to requesting nodes. Reducing these overheads is crucial for further improvement of the remote I/O throughput. However, the *clustering* policies exhibit good replication throughput performance than the *no_clusting* policy; we see that the *proposal*-1024 policy is 1.46 times and the *proposal*-5120 policy is 1.58 times efficient than the *no_clustering* policy. This indicates that the clustering based replication technique can achieve efficient data management, which places replicas in optimal locations, while minimizing the influence of data transfer over networks.

The overheads of the replication decision are mainly derived from the time to solve ILP problems, which are related to the performance of the solver program; we use GNU GLPK [2] in our prototype. We evaluate the performance of the same determining replica placement problem using GLPK and CPLEX[4] that is the fastest but commercial solver program. We determine ten file cluster placements in the *cluster*-5120 policy using a single node, which consists of Xeon X5460 (3.16 GHz) with 48 GB of memory, running Linux 2.6.18. We observe that GLPK takes 356 sec, while CPLEX takes 4.55 sec to solve the above problems; further improvements are required to support many file clusters in a large-scale grid environment with complex workloads, such as employing further efficient ILP solvers.

6. Conclusion

We have presented a file clustering based replication algorithm for grid file systems, which groups files stored in a grid file system according to the relationship of simultaneous file accesses and determines locations and movement of replicas of file clusters from the observed performance data of file accesses. Our experiments on a given grid environment suggests that the proposed algorithm achieves accurate file clustering and efficient replica management.

In the future, we will continue to evaluate our algorithm

Table 3. Performances of Each Data Management Policy

	io thput (MB/s)	st size (MB)	rep thput (MB/s)	dl time (sec)	dm time (sec)
no_rep	1.71	9761	N/A	N/A	N/A
pid-1024	4.63	16348	7.19	1590	21
pid-5120	4.06	15682	7.80	3134	27
no_cluster	4.89	17689	4.94	888	97

in complex workloads on live environments. In particular, reducing overheads such as determining replication and actual replica transfer is crucial for efficient data management in a large-scale grid environment. The next step is to combine our prototype with other efficient ILP solvers and efficient replica transfer techniques.

Acknowledgements

This research is supported in part by the MEXT Grant-in-Aid for Scientific Research on Priority Areas 18049028 and the JSPS Global COE program entitled "Computationism as Foundation for the Sciences".

References

[1] Gfarm. http://datafarm.apgrid.org.
[2] GLPK (GNU Linear Programming Kit). http://www.gnu.org/software/glpk.
[3] Hadoop. http://hadoop.apache.org/.
[4] ILOG CPLEX. http://www.ilog.com/products/cplex/.
[5] InTrigger. http://www.intrigger.jp.
[6] METIS. http://glaros.dtc.umn.edu/gkhome/views/metis.
[7] NCBI BLAST. http://www.ncbi.nlm.nih.gov/BLAST.
[8] The DZero Experiment. http://www-d0.fnal.gov.
[9] XtreemFS. http://www.xtreemfs.com.
[10] A. Amer, D. D. E. Long, and R. C. Burns. Group-based management of distributed file caches. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 525–534, 2002.
[11] J. Bent, D. Thain, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Explicit control in a batch aware distributed file system. In *Proceedings of the 1st USENIX/ACM Conference on Networked Systems Design and Implementation*, San Francisco, CA, March 2004.
[12] B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (nfs) version 4 protocol, April 2003. RFC 3530.
[13] S. Doraimani and A. Iamnitchi. File grouping for scientific data management: lessons from experimenting with real traces. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 153–164, 2008.
[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 96–108, Bolton Landing, NewYork, October 2003.
[15] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.
[16] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, volume 31, pages 264 – 275, December 1997.
[17] Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings of th 11th IEEE International Symposium on High Performance Distributed Computing*, pages 352–358, 2002.
[18] K. B. Rashedur M. Rahman and R. Alhajj. Study of different replica placement and maintenance strategies in data grid. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 171–178, 2007.
[19] H. Sato, S. Matsuoka, T. Endo, and N. Maruyama. Access-pattern and bandwidth aware file replication algorithm in a grid environment. In *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing*, pages 250–257, Tsukuba, September 2008.
[20] S. Shankar and D. J. DeWitt. Data driven workflow planning in cluster management systems. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing*, pages 127–136, New York, NY, USA, 2007.
[21] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid datafarm architecture for petascale data intensive computing. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 102 – 110, 2002.
[22] S. Venugopal, R. Buyya, and K. Ramamohanarao. A taxonomy of data grids for distributed data sharing, management, and processing. *ACM Computing Surveys*, 38(1):3, 2006.
[23] Y. Wang and D. Kaeli. Load balancing using grid-based peer-to-peer parallel I/O. In *In Proceedings of IEEE International Conference on Cluster Computing*, pages 1 – 10, 2005.