Adaptive Replica Placement in Hierarchical Data Grids

This content has been downloaded from IOPscience. Please scroll down to see the full text.
2010 J. Phys.: Conf. Ser. 256 012020
(http://iopscience.iop.org/1742-6596/256/1/012020)

View the table of contents for this issue, or go to the journal homepage for more

Download details:

IP Address: 52.27.110.214
This content was downloaded on 11/07/2015 at 00:48

Please note that terms and conditions apply.
Adaptive Replica Placement in Hierarchical Data Grids

Mohammad Shorfuzzaman, Peter Graham and Rasit Eskicioglu
Department of Computer Science, University of Manitoba, Winnipeg, MB R3T 2N2
E-mail: {szaman, pgraham, rasit}@cs.umanitoba.ca

Abstract. Data grids support distributed data-intensive applications that need to access massive (multi-terabyte or larger) datasets stored around the world. Ensuring efficient and fast access to such widely distributed datasets is hindered by the high latencies of wide-area networks. To speed up access, data files can be replicated so users can access nearby copies. Replication also provides high data availability, decreased bandwidth consumption, increased fault tolerance, and improved scalability. Since a grid environment is highly dynamic, resource availability, network latency, and users requests may change frequently. To address these issues a dynamic replica placement strategy that adapts to dynamic behavior in data grids is needed.

In this paper, we extend our earlier work on popularity-based replica placement proposing a new adaptive algorithm for use in large-scale hierarchical data grids. Our algorithm dynamically adapts the frequency and degree of replication based on data access arrival rate and available storage capacities. We evaluate our algorithm using OptorSim. Our results show that our algorithm can shorten job execution time greatly and reduce bandwidth consumption compared to its non-adaptive counterpart which outperforms other existing replica placement methods.

1. Introduction

Grid Computing [1] enables the selection, sharing, and aggregation of a wide variety of geographically distributed resources for solving large-scale computational problems. Data grids [2] provide services and infrastructure for distributed data-intensive applications that need to access massive datasets stored across distributed storage sites. An example of this is the experiments being run at the European Center for Nuclear Research (CERN) using the Large Hadron Collider (LHC) [3]. The CMS [4] and ATLAS [5] experiments using the LHC will collect massive amounts of data (many petabytes per annum) and involve thousands of researchers around the world. In such environments, maintaining a local copy of data on each accessing site is cost prohibitive. Further, storing such amounts of data centrally is impractical due to remote access latency and reliability concerns. Given the characteristics of the wide-area networks underlying many Grid systems and the need to access large volumes of data, scalability, data availability and access speed are key issues.

One way to speed up access in data grids is to replicate datasets at multiple locations [2]. Data replication not only reduces access costs, but also increases data availability [6, 7]. Creating replicas allows the routing of client requests to different replica sites thereby distributing the workload across the replica servers. The network load is also distributed across multiple network paths thereby decreasing the probability of congestion-related performance degradation.
To maximize the possible gains from file replication, strategic placement of the file replicas is critical [8, 9, 10]. Replication methods can be classified as static or dynamic [11]. With static replication, after a replica is created, it will be stored in the same location until it is deleted. Static replication cannot adapt to changes in file access patterns so the benefits of replication may be limited. On the contrary, dynamic replication automatically creates new replicas for frequently referenced data files or moves the replicas to other sites, as needed, to improve performance.

This paper addresses the problem of replica placement in data grids using an adaptive dynamic placement algorithm that extends our earlier work on Popularity-Based Replica Placement (PBRP) [12] which selectively replicates files based on their popularity. Our adaptive version, APBRP, presented in this paper dynamically adapts the frequency and degree of replication based on request arrival rates from clients and the available storage at replica servers to provide reduced access time and efficient use of bandwidth and storage resources. To evaluate the performance of APBRP, we use the data grid simulator OptorSim [13] and a hierarchical data grid structure reflecting the characteristics of the Compact Muon Spectrometer (CMS) experiments [4] at CERN. Simulations are done varying replica server capacities and data access patterns. We compare APBRP to PBRP and, transitively, to other replica placement algorithms.

The rest of this paper is organized as follows. Section 2 describes the hierarchical data grid model considered. In Section 3, research on replica placement in data grids is reviewed. We present our new algorithm in Section 4. Our simulation methods and performance results are described in Sections 5 and 6, respectively. Finally, Section 7 concludes the paper and suggests some directions for future work.

2. Hierarchical Data Grids
As is common in the current literature [14, 3, 15, 16], we focus on hierarchical data grids (e.g. [15]). Our example hierarchical data grid is modelled on the Large Hadron Collider (LHC) Computing Grid (LCG) [3] which provides data storage and analysis infrastructure for the thousands of users worldwide using the LHC. The data from the LHC experiments, estimated at roughly 15 Petabytes annually, is organized using a four-tier hierarchy with CERN as the root, or “tier 0” site, which stores all data produced. After initial processing, this data is distributed to the regional tier-1 sites that make data available to many national tier-2 sites. Individual scientists will access data through tier-3 sites (e.g. a local cluster in a University Department). The tier-3 sites is where users request access to their required data.

To minimize data access time and network load, replicas can be placed from the root to regional, national, or institutional centers. Thus, a user at a local site will try to access a replica locally. If a replica not present, the request will go to the parent node to find a replica there.
Generally, a user request goes up the hierarchy and uses the first replica encountered along the path to the root. For example, Figure 1 shows a four-tiered hierarchical data grid. Assume a user at node \( k \) tries to access a data file required by the job running on \( k \). The data cannot be found locally, so the request goes to the parent node \( g \), where the data is not available either. Finally, the request reaches node \( c \), and is served there.

3. Related Work
Ranganathan and Foster ([15, 16]) describe and evaluate various replication strategies for hierarchical data grids. These strategies are defined depending on when, where, and how replicas are created and destroyed. They compare six different replication strategies: 1) No Replication: only the root node holds replicas; 2) Best Client: replica is created for the client who accesses the file the most; 3) Cascading: a replica is created somewhere on the path from the root node to the best client; 4) Plain Caching: local copies are stored on initial request; 5) Caching plus Cascading: combines plain caching and cascading; 6) Fast Spread: file copies are stored at each node on the path from the root to the best client. Results show that the cascading strategy reduces response time by 30% over plain caching when data access patterns contain both temporal and geographical locality. When access patterns contain some locality, Fast Spread saves significant bandwidth over other strategies.

Park et al. [17] propose a dynamic replication strategy, called BHR (Bandwidth Hierarchy based Replication), to reduce data access time by avoiding network congestion. BHR accesses replicas located at sites that have the best current bandwidth to a job’s execution site. The BHR strategy can thus exploit the ‘network-level locality’ that occurs naturally in regional (and other) networks.

Two dynamic replication mechanisms are proposed by Tang et al [11] for multi-tier data grids. The Simple Bottom-Up (SBU) algorithm replicates any data file that currently exceeds a pre-defined threshold of access rate as close as possible to the clients without considering historical accesses. The Aggregate Bottom-Up (ABU) algorithm considers the access histories of files used by sibling nodes and aggregates the access information for files so that the frequently accessed files are replicated first. This process is repeated from the leaves to the root. ABU improves access latency significantly at the expense of increased storage use.

Different sites in a data grid may have different service quality requirements making Quality of Service (QoS) another important factor. Wang et al. [18] describe a heuristic QoS-aware algorithm that places replicas to simultaneously improve system performance and satisfy users’ quality requirements. Their algorithm, for general graphs, starts by finding the cover set of every server in the network. The algorithm then identifies and deletes super cover sets in the network. Finally, it inserts replicas into the network iteratively until all servers are satisfied. Results indicate that their algorithm finds near-optimal solutions.

In our earlier work [12], we presented the Popularity Based Replica Placement (PBRP) algorithm for hierarchical data grids. Since data access patterns may change over time, any dynamic replication strategy must track file access histories to decide when, what and where to replicate. The “popularity” of a file is determined by its access rate at each location and PBRP is invoked at regular intervals to create replicas as close as possible to those clients that frequently request data files. Replication is done in two phases. The bottom-up aggregation phase aggregates access history records for each file to upper tiers (as in ABU) summing the access counts for records whose nodes are siblings and which refer to the same file. In the second, top-down, phase the aggregated information is used to place replicas. The placement process traverses down the hierarchy as long as the aggregated access count is greater than or equal to a pre-defined popularity threshold (selected empirically based on historical access information). A replica is placed at a node if the threshold prevents further traversal through one or more of its children. An instance of bottom-up aggregation and top-down placement is shown in Figure 2.
Our simulation results show that PBRP can shorten job execution time significantly and reduce bandwidth consumption compared to other dynamic replication methods including ABU, Fast Spread and Cascading placement. However, the effectiveness of PBRP depends on careful selection of the popularity threshold. The adaptive algorithm proposed in this paper is designed to address this issue by determining the threshold value dynamically based on the data access arrival rates and available storage at replica servers.

4. Adaptive Replica Placement
The goal of our Adaptive Popularity Based Replica Placement (APBRP) algorithm is to reduce data access time by dynamically creating replicas for “popular” data files. Placement of the replicas is guided by dynamic adjustment of the threshold value used in PBRP that determines how deep and hence how often files are replicated in different parts of the hierarchy. APBRP sets the threshold dynamically based on client request rate and replica server capacities.

Like PBRP, replication in APBRP is done in two phases. The first phase aggregates access records for each file from lower to upper tiers. In the second phase, replicas are placed from the root to the leaves of the tree using the aggregated information and threshold value. As in PBRP, our algorithm is invoked at regular intervals. The access pattern logs are cleared at the start of each interval to capture evolving pattern changes. A short interval can be chosen for high arrival rates incurring greater overhead but adapting more rapidly to changing access patterns. Determination of the threshold is done independently of placement.

We use access request rate and available server capacity to determine the threshold value. A high access request rate corresponds to frequent accesses by clients which, in PBRP, results in more “popular” files whose access counts exceed the threshold. This in turn results in more replicas which causes additional replication overhead (in terms of both bandwidth and storage use). Increasing the threshold lessens the number of replicas created. On the other hand, when the request rate drops, the number of popular files is less so fewer replicas are created even though the system might be capable (in terms of bandwidth and storage) of supporting more replicas to improve access latency. The available storage capacities of replica servers are also used in adapting the threshold value. The required threshold value can be decreased to create more replicas if replica servers have abundant storage space. The threshold value can also be increased to lessen the number of replicas as the replica servers become full.

Figure 3 illustrates replica placement with dynamic threshold adjustment. First, the variation in request access rate from the users is calculated to determine the change in threshold value.
Figure 3. Replication with threshold adjustment

(discussed later). Then, the replica placement algorithm uses the new threshold value to determine the replica placement. A storage checker assesses the storage used by all replica servers to decide whether any further adjustment in threshold value is necessary. Feedback is sent to the threshold controller in case of heavily loaded or under-utilized replica servers.

The key challenge in this approach is to determine how much change in the threshold value the system can tolerate based on changes in request arrival rate and available storage capacities of replica servers. Overly aggressive threshold increases will cause the number of replicas to drop which will increase access time and decrease storage utilization. Aggressive threshold decreases will increase the number of replicas and the storage cost. Hence, the threshold value has to be adjusted so that the trade-off between space use and access latency is balanced.

4.1. Determining the Initial Threshold
The initial threshold value is set based on the average aggregated access counts at the replica servers in the bottom tier. The value of the average aggregated access count is calculated by dividing the total number of aggregated access counts for a file at the replica servers in the second to lowest tier of the hierarchy by the number of replica servers at that tier. This is done during the bottom-up aggregation phase. The initial value is then adjusted dynamically based on available storage and user request arrival rates.

4.2. Dynamic Adjustment of the Threshold Value
Changes to the threshold based on variation in the request arrival rate are not applied immediately to avoid reacting to transient changes in the arrival rate. Changes are only made to reflect an ongoing trend over an interval of time. This interval is calculated as a fraction of the length used for sampling the access histories of clients. The threshold value is increased or decreased by the difference between the current and previous average aggregated access counts of replica servers in the lower tier of the hierarchy. For example, assuming that $R_i$ is the new average aggregated access rate in the $i^{th}$ interval and $R_{i-1}$ is the average aggregated access rate in the $(i-1)^{th}$ interval, the threshold value for the $i^{th}$ sampling interval, $threshold_i$, is calculated as follows:

$$threshold_i = threshold_{i-1} + \Delta$$

where $\Delta$ is defined as:

$$\Delta = \begin{cases} 0, & \text{for the first interval of access histories} \\ R_i - R_{i-1}, & \text{otherwise} \end{cases}$$

Once the threshold value is updated, the available storage capacities of the replica servers are checked. Further adjustment of the threshold value is done based on the available capacities.
### Figure 4.

(a) The simulated data grid topology, (b) Different storage configurations

At the replica servers. The threshold controller (Figure 3) updates the threshold based on the ratio between the desired percentage of storage use and the actual percentage of storage used by replica servers. The new threshold value for the $i^{th}$ interval is thus calculated as:

$$\text{threshold}_{i}(\text{updated}) = \text{threshold}_{i} \times \text{ratio}$$

In this paper, we define three storage utilization states for when the system is lightly, moderately, and highly loaded corresponding to the percentage of storage capacity used by replica servers (less than 50%, 50% to 75%, and greater than 75%, respectively). If the system is heavily loaded or under utilized then the calculated ratio is used in updating the threshold.

We also use a popularity-based form of the Least Recently Used (LRU) replacement policy with an added constraint to ensure that replicas created in the current replication interval will not be replaced. This additional constraint is needed to avoid the deletion of newly created replicas by the dynamic replication algorithms. If the available space on a given server is less than the size of the new replica, some “removable” replicas need to be deleted to make room for the new replica. Removable replicas are those created before the current replication interval that are not in use by any client at present.

### 5. Simulation Setup

To evaluate APBRP we used a locally extended version of OptorSim [19] created to support the necessary information gathering needed for the ABPR algorithm.

#### 5.1. Simulated Data Grid Topology

The simulated data grid structure and network bandwidths between sites are based on estimates for the CMS experiment. The topology of the simulated data grid is shown in Figure 4. All data requests are generated by the leaf nodes. The links between nodes show the available network bandwidth. In our simulations, we considered five different storage configurations based on the relative storage capacity of the replica servers, defined as a ratio between the total size of the replica servers ($S$) and the total size of all data files in the system ($F$). There are 2500 data
files in the system and each one is 10 GB, so $F$ is approximately 25 TB. The relative storage capacity of the replica servers in the data grid are varied from 75% down to 13%. The detailed storage configurations for all five cases are shown in Figure 4. For example, in configuration 2, each node in tier-1, tier-2, and tier-3 has 500 GB, 200 GB, and 50 GB storage available for replicas, respectively. The absolute capacities are of little interest but the relative capacities affect placement decisions. The total storage size of the replica servers is 13.75 TB and the relative storage capacity is 55% ($\frac{13.75}{25} \times 100$). To assess the impact of file size distribution on the performance of the replication algorithms, we also considered file sizes of 2 GB and 20 GB. We adjusted the number of files so the total size of all data files in the system and the relative storage capacity of the servers were unchanged.

5.2. Data Access Patterns

Each simulated job requests a set of filename(s) to access. The order in which the files are requested is determined by a selected access pattern. We experimented with five access patterns: sequential (files are accessed in the order stated in the job configuration file), random (files are accessed using a flat random distribution), unitary random (file requests are one element away from previous file request but the direction is random), Gaussian random walk (files are accessed using a Gaussian distribution), and the heavily tailed Zipf distribution: $P_i = \frac{K}{r}$, where $P_i$ is the frequency of the $i$th ranked item, $K$ is the popularity of the most frequently accessed data item and $s$ determines the shape of the distribution.

We ran all simulations with 150 jobs of various types. The jobs were submitted with a fixed probability but some jobs were more common than others. A resource broker acted as a meta-scheduler that controlled job scheduling to different client nodes with computing resources based on the estimated data access cost for the current and all queued jobs.

6. Simulation Results

The metrics we chose to evaluate the performance of APBRP were job run time, average bandwidth use, and storage use. Run time is the total time to execute all jobs and is scaled for display. For each data transmission, the bandwidth cost is the data size multiplied by the summation of costs along the data transmission path. The value of the average bandwidth cost is the total bandwidth cost divided by the number of client requests. Storage use is the percentage of the available storage used by the replicas. To determine the effectiveness of our adaptive algorithm we considered three scenarios: when the data access rate consistently increases, consistently decreases, and when it fluctuates. For each case, we compared APBRP with its non-adaptive counterpart PBRP for the various replica server configurations.

![Figure 5. Runtime: Config. 1, Fluctuating Rate](image-url)
6.1. Job Execution Time
Figure 5 shows the job execution times of APBRP and PBRP using configuration one (uniform capacities) when the data access arrival rate from the clients is fluctuating. APBRP displays shorter execution times for all data access patterns because APBRP adjusts the threshold value based on the varying access arrival rate which leads to the creation of more replicas which decreases the run time. For example, with a Zipf distribution, the number of replicas created by APBRP and PBRP were 37 and 27, respectively. The benefit of APBRP is most significant when the accesses exhibit some randomness. APBRP clearly adapts well to fluctuating accesses.
access rate fluctuates. With decreasing storage size, the execution times of both techniques are increased but by different amounts. APBRP shows shorter run times for all storage configurations. However, when the available capacity is smaller, the benefit of APBRP over PBRP decreases since there is less available space to allocate additional replicas.

Figure 8 shows the job times of APBRP and PBRP for all storage configurations using the Zipf access pattern with file sizes of 2 GB and 20 GB. In both cases, APBRP displays shorter execution times for all configurations. Unlike the other tested scenarios, the benefit of APBRP over PBRP increases with decreasing storage size when a file size of 20 GB is used. This is due to the increased number of replicas and relatively low replica replacement frequency in APBRP.

Figure 9 compares the run times of APBRP using different file sizes for the Zipf distribution. The execution time with a file size of 20 GB is much higher than the other two cases due to the increased data transfer latency and frequent replacement of new replicas. The performance of the replication algorithm in terms of job execution time is not much impacted when smaller file sizes (2 GB and 10 GB) are used. The job times are quite similar for both cases except for some storage configurations where the use of a 2 GB file size results in shorter execution times.

Figure 10 shows the run times of APBRP and PBRP, again for configuration one, as the client request arrival rate decreases. APBRP shows consistently better performance for all data access patterns. This is due to the fact that when the arrival rate decreases consistently, APBRP also decreases the threshold which results in more replicas compared to PBRP. This in turn decreases the execution time. The Zipf distribution shows the lowest run time using APBRP.
Figure 10. Runtime: Config. 1, Decreasing Rate

Figure 11 compares the job execution times of the two placement algorithms using different storage resource configurations as the access rate decreases consistently. With decreasing capacity, the execution times for both algorithms increase but by different degrees. APBRP shows better job execution times for all storage configurations. However, when the available storage capacity is smaller, the benefit of APBRP over PBRP decreases as before. This happens because of frequent replica replacement in APBRP when the replica servers have limited storage.

Figure 12 compares the run times of APBRP and PBRP, again for configuration one, as the request arrival rate increases. When the request arrival rate increases, APBRP increases the threshold to limit the number of replicas to a reasonable level based on the storage available (as
described in section 4.2) while PBRP experiences a drastic increase in the number of replicas (that eventually fills the replica servers).

APBRP performs as well as PBRP or somewhat better for the Zipf, Unitary, and Sequential access patterns but with the Random and Gaussian distributions PBRP performs better due to the significant drop in the number of replicas when using ABRP. For example, PBRP creates approximately 66 and 56 replicas for these two access patterns. These numbers drop to 29 and 25 with APBRP. Overall, APBRP shows less benefit over PBRP, for the tested scenarios, when the client request rate is consistently increasing.

Figure 13. Runtime by Pattern & Config, Increasing Rate

Figure 13 compares the job execution times of the two placement algorithms using different
storage resource configurations with consistent increase in access rate. With decreasing capacity, the job execution times using both algorithms increase but by different amounts. Adaptive-PBRP maintains better job execution times for all five storage configurations. For the Zipf and Random patterns, the performance difference between the methods reduces as the storage capacities of the servers decrease. This happens because the difference between the number of replicas created by PBRP and Adaptive-PBRP decreases significantly.

**Figure 14.** Avg. B/W: Config. 1, Fluctuating Rate

**Figure 15.** B/W by Pattern & Config, Fluctuating Rate
6.2. Average bandwidth cost

Figure 14 shows the average bandwidth (B/W) costs of APBRP and PBRP for configuration one as the request access arrival rate fluctuates. APBRP requires less bandwidth for all file access patterns except sequential. ABRP’s increased number of replicas close to clients compared to PBRP generally results in decreased bandwidth consumption for accessing files.

![Average bandwidth cost](image)

**Figure 16.** Avg. B/W by File Size, Fluctuating Rate

The average bandwidth costs for all storage configurations and access patterns given regular fluctuation in request rates are shown in Figure 15. With decreasing storage size, the bandwidth costs of both techniques are increased but by different amounts. APBRP shows lower bandwidth costs for almost all storage configurations and access patterns but the benefit of APBRP over PBRP decreases as the storage capacities of replica servers are reduced.

![Average bandwidth cost](image)

**Figure 17.** Avg. B/W: APBRP by File Size, Fluctuating Rate

Figure 16 shows the average bandwidth costs of APBRP and PBRP for all storage configurations using the Zipf access pattern with file sizes of 2 GB and 20 GB. For both cases, APBRP has less bandwidth cost for all but one configuration. The benefit of APBRP over PBRP decreases somewhat with decreasing available storage. Since there is less available space for storing more replicas, the frequency of replica replacement increases which in turn leads to additional bandwidth consumption.

Figure 17 compares the average bandwidth costs of APBRP using different file sizes for the Zipf distribution. The bandwidth cost for 20 GB files is much higher than the other two cases due to the increased data transfer cost and the frequent replacement of new replicas.
The performance of the replication algorithm in terms of average bandwidth consumed is not significantly impacted for smaller file sizes (2 GB and 10 GB). The bandwidth costs are similar in both cases. However, 2 GB files result in less bandwidth consumption than 10 GB files.

![Figure 18. Avg. B/W: Config. 1, Decreasing Rate](image)

Figure 18 shows the bandwidth costs for storage configuration one when the request arrival rate is decreasing. APBRP shows better performance for only some access patterns. The benefit of APBRP over PBRP is not terribly significant and the liability for sequential access is great.

![Figure 19. Avg. B/W by Pattern & Config, Decreasing Rate](image)

Figure 19 shows the average bandwidth costs of both strategies for the three configurations with consistent decrease in access rate. Adaptive-PBRP again requires less bandwidth compared to PBRP. As the storage size decreases, the bandwidth costs of both the strategies increase.
Figure 20. Config 1., Avg. B/W: Increasing Rate

Figure 20 shows the bandwidth costs for configuration one as the request arrival rate from the clients increases. As the request arrival rate increases, PBRP experiences a drastic increase in the number of replicas. This in turn results in consumption of extra bandwidth for unnecessary data movement. APBRP adjusts the threshold value to reduce the number of replicas and thus saves significant link bandwidth. For all data access patterns, APBRP requires less bandwidth than PBRP.

6.3. Storage Use
Figure 21 shows the storage use of APBRP and PBRP using server configuration one as the request arrival rate fluctuates. APBRP requires more storage than PBRP for all data access patterns. This is due to the creation of an increased number of replicas using APBRP.

Figure 21. Storage Use: Config. 1, Fluctuating Rate

Figure 22 compares the storage use of the two placement algorithms for all storage configurations and access patterns as the request rate fluctuates. With decreasing storage availability, the percentages of storage used for both techniques increases but by different amounts. When the available storage capacity is limited, the difference in storage use for APBRP and PBRP tends to decrease. This happens because of frequent replica replacement in APBRP since the replica servers have limited storage available.
Figure 22. Storage Use by Pattern & Config, Fluctuating Rate

Figure 23. Storage Use by File Size, Fluctuating Rate

Figure 23 shows the storage use of APBRP and PBRP for all storage configurations and the Zipf file access pattern with file sizes of 2 GB and 20 GB. In both cases, APBRP has greater storage cost due to its creation of additional replicas when compared to PBRP.

Figure 24 compares the storage costs of APBRP using different file sizes for the Zipf distribution. The performance of APBRP in terms of storage cost is not much impacted by the data file size. Storage use is very similar for all three cases and storage configurations.

Figure 25 (left) shows the storage use of APBRP and PBRP for configuration one as the request arrival rate decreases. When this happens, APBRP decreases the threshold value which results in an increased number of replicas compared to PBRP. This increases the storage use.
Figure 24. Storage Use: APBRP by File Size, Fluctuating Rate

Figure 25. Storage Use: Config. 1, Decreasing (left) and Increasing Rate (right).

Figure 25 (right) shows the percentage of storage used by APBRP and PBRP for configuration one as the request arrival rate increases. In PBRP, the number of replicas increases drastically with the increase in request arrival rate whereas APBRP limits the number of replicas to a reasonable level by increasing the threshold value resulting in lower storage costs than PBRP.

7. Conclusions and Future Work
In this paper, we have introduced an adaptive popularity-based replica placement algorithm (APBRP). Simulation results show that our algorithm can significantly shorten job execution time and reduce bandwidth consumption at the cost of moderate increases in storage use compared to PBRP. For the tested access patterns and storage configurations, APBRP outperforms PBRP in terms of both job execution time and average bandwidth use. The increased consumption of (relatively cheap) storage space by APBRP makes it a good candidate when access time and bandwidth use are of primary concern. Our earlier results [12] show that PBRP can shorten job execution time significantly and reduce bandwidth consumption compared to other dynamic methods including ABU, Fast Spread and Cascading placement. Thus, transitively, APBRP also performs better than these other non-adaptive dynamic replication methods. The novelty of APBRP is its ability to balance space use and access latency by means of adaptive popularity-based file replication. The improved efficiency of access to data and flexibility to changing access patterns offered by APBRP will help make large-scale data intensive applications practical, potentially running over public wide-area networks.
For future work, we are exploring the incorporation of user QoS requirements into the placement process. We also plan to explore how to exploit network knowledge in making placement decisions. Finally, we will investigate dynamic replica maintenance (e.g. replica consistency) and work to extend our approach to non-hierarchical network models.

Acknowledgments
This research was supported in part by the Natural Sciences and Engineering Research Council of Canada and by a University of Manitoba Graduate Fellowship.

References
[1] Kesselman C and Foster I 1998 The Grid: Blueprint for a New Computing Infrastructure (Morgan Kaufmann Publishers)
[2] Venugopal S, Buyya R and Ramamohanarao K 2006 ACM Comp. Surveys 1(38) 1–53
[3] Worldwide LHC Computing Grid http://lcg.web.cern.ch/lcg/ Distributed Production Environment for Physics Data Processing
[4] Holtman K 2001 CMS Data Grid System Overview and Requirements CMS Experiment Note 2001/037, CERN
[5] The ATLAS experiment http://atlas.ch/ Particle Physics Experiment at CERN
[6] Ranganathan K, Iamnitchi A and Foster I 2002 Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID’02) pp 376–381
[7] Lamehamedi H, Szymanski B, shentu Z and Deelman E 2002 Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing pp 378–383
[8] Wolfson O and Milo A 1991 ACM Transactions on Database Systems 16 181–205
[9] Kalpakis K, Dasgupta K and Wolfson I 2002 Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID’02) pp 376–381
[10] Tu M, Li P, Ma Q, Yen I and Bastani F 2005 Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS’05) vol 1 p 14
[11] Tang M, Lee B, Yeo C and Tang X 2005 Future Generation Computing System 21 775–790
[12] Shorfuzzaman M, Graham P and Eskicioglu R 2008 Proc. 9th Intl. Conf. on Parallel & Dist. Computing, Applications and Technologies pp 524–531
[13] Bell W, Cameron D, Capozza L, Millar A, Stockinger K and Zini F 2002 Proc. 3rd Intl. IEEE Workshop on Grid Computing (Grid’2002)
[14] Bell W H, Cameron D G, Carvajal-Schiaffino R, Millar A P, Stockinger K and Zini F 2003 Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid
[15] Ranganathan K and Foster I T 2001 Proc. Intl. Wksp. on Grid Computing pp 75–86
[16] Ranganathan K and Foster I 2001 Proc. of the Intl. Conf. Computing in High Energy and Nuclear Physics (2001).
[17] Park S, Kim J, Ko Y and Yoon W 2003 Proc. 2nd Intl. Workshop on Grid and Cooperative Computing
[18] Wang H, Liu P and Wu J 2006 Journal of Grid Computing 96–103
[19] Bell W, Cameron D, Capozza L, Millar P, Stockinger K and Zini F 2003 Intl. Journal of High Performance Computing Applications 17(4) 403–416