Efficient Distributed Disk Caching in Data Grid Management *

Song Jiang and Xiaodong Zhang

Department of Computer Science,
College of William and Mary
Williamsburg, VA 23187, USA

Abstract

Effectively utilizing disk caches is critical for delivering and sharing data in data-grids considering the large sizes of requested files and excessively prolonged file transmission time. An essential component in the disk cache management is its replacement policy that determines which file(s) are least valuable and should be evicted to create space for incoming files. Though a large number of replacement algorithms for data objects of different sizes have been proposed recently in the domain of Web-caching and disk caching in data grids, they inherit the shortcomings of the LRU and LFU replacements in characterizing access patterns. In order to address this limit, we propose a technique to measure relative file access locality strength – how soon a file is to be re-accessed before being evicted compared with other files. When we estimate the in-cache re-access probability, we take the disk space consumed by accessed files as well as disk cache size into consideration. Using a relative locality strength estimation, we are able to accurately rank the value of each file for being cached, and select the file(s) with least values for replacement.

Our simulation results show that our proposed policy is the most effective one among existing policies in interpreting access patterns, and consistently achieves performance improvement measured by hit ratios and byte hit ratios.

1. Introduction

Every year a huge amount of scientific data are generated by simulations or collected from large scale experiments and used by researchers around the world. While accesses to these data in data grids become the main bottleneck in data-intensive applications, a middleware called storage resource manager (SRM) in data grids is developed to address the issue. SRM manages large capacity disk caches in a dynamic fashion for caching frequently requested files and providing fast accesses for clients. The disk caches are located between clients and source data storage. There are several reasons for disk cache to be an essential component of data grids: (1) Majority of data files reside in mass storage system (MSS) including tertiary storage system such as robotic tapes, high performance storage systems (HPSS), and RAID farms. It takes long latency (up to several minutes) to retrieve these data at their sources; (2) It takes a very long time (up to a few hours) to complete file transfers for a request over wide-area networks; (3) A researcher's workstation or even her local computer center may not be able to keep all the required dataset for a long time for her needs. With disk caches that temporarily store a set of frequently accessed files, a client can access its desired files much faster from the caches than from MSS systems.

File replacement algorithms in SRMs determine which files can be kept in the cache for re-use, and which files are less likely to be accessed and should be replaced. Considering the large sizes of caching files and the excessively prolonged file transmission time, a replacement algorithm has a significant influence on the efficiency of data accesses. There are several factors that have to be accounted in devising a replacement algorithm for objects with varied sizes and transfer cost, namely, (1) file access pattern, (2) file size, and (3) file transfer cost.

An effective replacement algorithm works because of existence of locality. It strives to abstract locality information from access patterns observed on-line. We define locality strength of an object as the inverse of the time to its next reference. It is known that an optimal algorithm chooses the object with the smallest

* This work is supported in part by the U.S. National Science Foundation under grants CCR-0098055 and ACI-0129883

locality strength for eviction with fixed sizes and an uniform miss penalty. For caching objects with varied sizes and a non-uniform miss penalty, we combine locality strength of a file with its size and miss penalty in an utility function to determine its benefit of being kept in the disk. For a given file i at time t, let $L_i(t)$ denote its locality strength, S_i denote its size, and C_i denote its retrieving cost (miss penalty), the utility function is

$$\phi_i(t) = L_i(t) * \frac{C_i}{S_i}.$$

This utility function framework is straightforward, and is broadly accepted in various replacement policies. Furthermore, it is not hard to know file sizes and to estimate their retrieving costs. However, our defined locality strength is an ideal, off-line measure of the benefit of being cached. The most challenging issue in implementing the function is how to estimate locality strength on-line by correctly interpreting history access patterns. In the three factors of an utility function, locality strength is the most critical one, which determines cache hit ratio, while the other two factors are mainly related to the amount of gain from the hits such as responsive time reduction and network bandwidth savings.

Otoo et al [8] recently proposed a disk cache replacement policy for data grids, called Least Cost Beneficial Based on the K backward References (LCB-K), where the utility function $\phi_i(t)$ is used, and their locality strength is estimated as follows:

$$L_i(t) = \frac{k_i(t)}{(t - t_{-k_i})} * g_i(t),$$

where for each file i, k_i is the number of the most recent references retained, up to a maximum of K, within the time interval $[t - t_{k_i}]$. t_{k_i} is the time of the $k_i(t)$ backward reference, $g_i(t)$ is the cumulative count of references to the file over the active period of references to the file. Though LCB-K thoughtfully incorporates history information into their strength estimation, and detailed simulation in the SRM environment shows its performance improvement over some representative replacement algorithms in terms of a reduction of the average retrieval cost, its significant weakness is that it uses absolute wall clock time for the estimation, which could be irrelevant to the re-use probability estimation, and misguide the selection of victim files.

In this paper, we propose a new method to timely estimate file locality strength based on the principle of timescale relativity. In our method we count on the accumulated size of accessed files and disk cache size to measure the probability of in-cache file re-use, rather than using traditional measures such as absolute wall clock reference time, reference frequency, or number of accessed files. We show the effectiveness of our replacement policy incorporating the locality strength estimator through simulation on real Grid workload trace. The contributions of our work are threefold: (1) We have identified a critical weakness in the utility function for accurately ranking the caching values for accessed files; (2) We propose a locality strength estimator using more relevant access events to effective compare the relative re-reference probability among accessed files; (3) Our real workload trace simulation shows its effectiveness for data grid management.

2. Other Related Work

A lot of replacement policies for varied object sizes and miss penalties have been proposed in the Web-caching and data-grid management domains. We classify these work into two groups based on whether an utility-function is used.

Some early proposed policies were simply based on traditional paging replacement algorithms such as Least-Recently-Used (LRU) and Least-Frequently-Used (LFU) without using utility functions. For example, LRU-Threshold algorithm [2] is the same like LRU, except that documents larger than a specified threshold size are not cached. Log(Size)+LRU algorithm [2] replaces the object with the large log(size) and is the least recency used object among all the object with the same log(size). Pitkow/Recker algorithm [12] evicts the least-recently-used object, except if all objects are accessed today, in which case the largest one is evicted. Existing studies show that none of the simple policies can perform consistently better than others across various traces. The essential problem of these algorithms is that they can not effectively combine the observed access pattern with the retrieving cost and object sizes.

Another group of policies make replacement decisions by calculating an utility function for each object and evict the one with the least utility. The framework of the function is in the form:

$$\phi = LocalityStrengthEstimator * \frac{RetrivingCost}{FileSize}$$

As we have mentioned, file sizes and retrieving costs are easy to observe or estimate. The challenge of these group of policies are centered on the estimation of locality strength, where they mainly differ from each other. We list the estimator of major policies of the group as follows:

1. **Hybrid algorithm** [11] used $(n_p)^{W_n}$ as the estimator, where n_p is the number of times object p has been requested since it was brought into the cache. W_n is an constant.
2. **Lowest Relative Value (LRV)** [7] formulates the utility function based on an extensive empirical analysis of trace data. They analyzed the relationship between the probability of objects' re-use and the number of its previous request times and its sizes. Then the derived relationship is used to estimate locality strength. Because their method heavily relies on the web-traces they used, it is not a general replacement applicable to disk cache management in data-grid.
3. **GreedyDual-Size(GDS)** [4] is actually a generalization of LRU, which allows varied cost and object sizes. Its locality strength estimator for an object is essentially the inverse of the number of missed objects since its last access. To overcome the shortcoming of GDS overlooking history access information prior to the last access, GreedyDual-Size with Frequency (GDSF) [1] incorporates object frequency (the number of access times) into the estimator. However, using frequency can cause the "cache pollution" problem , where an object having accumulated a high frequency will stay in cache for excessive long time, even if it is not used any more.

3. Timescale Relativity Principle in Disk Caching

The locality strength estimation is about how often a file has been accessed and how soon it will be accessed. Both "how often" and "how soon" are related to the measurement of time. In this work, we strongly advocate to use the principle of timescale relativity in measuring the time used in the paging replacement algorithm design [6, 9]. We name the time used in caching algorithms *caching time*. The principle has two implications in disk caching design. The first is that only events that matter for replacement decisions should count to advance caching time. For example, the order of requests for retrieving files is important to predict the future file access sequence, and to determine which files would be accessed relatively less soon and should be replaced. However, if we use wall clock ticks to advance caching time like the one in LCB-K, the order would be obscured in the long wall-clock time of days even weeks. Actually caching time should only advance with file access events. The second implication is that the pace of caching time advance should be related to the sizes of accessed files. Time advances at a slower rate for accessing files with small sizes than accessing files with large sizes. That is to say, regardless of the amount of wall clock time elapsed or the number of files accessed, the probability of re-reference to the file before it is evicted from the cache becomes slim, if the total size of other files accessed after the last reference to a file becomes large. This is because large cache space would be consumed after its last access. That is, large total accessed file sizes mean a long caching time. However, this property of disk caching can not be captured in the GDS, LCB-K and other existing algorithms. Our proposed replacement policy addresses this critical weakness.

4. Our Disk Caching Policy: Least Value based on Caching Time (LVCT)

Our disk caching policy is also based on the evaluation of the utility function, in which we adopt a different locality strength estimator based on the timescale relativity principle.

4.1. Locality Strength Estimator

If there are n distinct, currently resident files f_1, f_2, ..., f_n accessed after last reference to file F, and the size of file f is $SIZE(f)$, the current caching time of file F is $\sum_{i=1}^{n} SIZE(f_i)$. In our definition, if file f_i is accessed multiple times before the reference to file F, its size is counted only once. This is because repeated references to the same file do not consume additional cache space, thus do not advance extra caching time. If a requested file is not admitted into disk cache and does not consume cache space, the caching time is not advanced. Our caching time definition considers the competition of cache space among accessed files, and expresses the eligibility for a file to be cached compared with other accessed files in terms of file access pattern. Holding the same principle of LRU replacement, we assume a file with a large caching time will not be accessed soon. The locality strength estimator of file F is the inverse of its current caching time. We use a data structure called caching time stack to keep track of caching time for each accessed file. Caching time stack is similar to the LRU stack used in LRU paging replacement algorithm. The difference from LRU stack is that its stack entry is a metadata representing a file, not a constant-sized page. Each entry records the current caching time of its corresponding file.

Figure 1 illustrates a caching time stack. When file f is accessed, we move it to the top of the stack, and set its caching time as 0. If the file was in the cache, we only advance the caching times of the files above

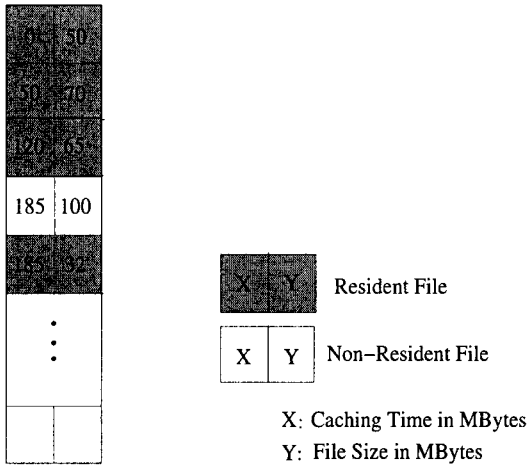

Figure 1. Caching time stack. Each entry in the stack represents an accessed file, and has two fields indicating its current caching time and the file size.

the original position of f by $SIZE(f)$; Otherwise, the caching times of all the files in the stack are advanced by $SIZE(f)$. The caching times of the files are dynamically maintained in the stack. Non-accessed files have infinite caching times. It is noted that some of entries of the stack could represent files which have been replaced out of the cache. To limit the size of the stack, we remove the entries in the bottom of the stack once either of the conditions are satisfied: (1) The total size of files represented in the stack exceeds two times of the disk cache size; or (2) The number of files represented in the cache exceeds two times of the number of resident files. The purpose of the caching time stack is to track the recently accessed files and to select most frequently ones which the disk cache can hold. When either of the conditions is met, the files associated with the entries in the stack bottom have so large caching times that they can be removed from the stack and be treated as non-accessed without negatively affecting the evaluation of locality strength of frequently accessed files.

4.2. LVCT Replacement Based on Caching Time

We define our utility function of file f_i as:

$$\phi_i(f_i) = \frac{1}{(CachingTime_i)} * \frac{cost_i}{size_i},$$

where $cost_i$ and $size_i$ are the retrieving cost and the size of file f_i, and $CachingTime_i$ is its current caching time, which is available from caching time stack. Then we have our replacement policy, called *Least Value Based on Caching Time* ($LVCT$):

For a request for file f, there are two cases:

1. If the file can be found in the cache, update the caching time stack, then transfer the file to the client;
2. If the file can not be found in the cache, then we send it to the source. When the retrieved file arrives, we select a set of file(s) with the least values based on our utility function to make enough room to accommodate the file. Then we compute the value of f based on the utility function. If the value of f is larger than that of any selected file in the set, we will replace the files and admit file f into the disk cache. Otherwise, file f will not be cached.

In our policy we overcome a common weakness in all previous replacement algorithms when they are applied in the disk cache of SRM. These algorithms do not evaluate the eligibility of missed files being cached in the disk. They always evict least valuable resident file(s) to make room for a missed file. Once the file is admitted into disk it could take a long time for the file to be evicted even if it would not be accessed any longer, because it is a recently accessed file and keeps its high utility ranking for a period of time as a resident file. In our proposed estimator, we allow non-resident files to keep their caching times in the stack for an extended period of time after the files are evicted from the cache. When the files are re-accessed, this additional history information can be used to compare its utility value against those of the resident files by a consistent criteria to decide whether they should be replaced.

5. Performance Evaluation

The metrics we use are hit ratio and byte hit ratio, which are two most popular performance measures in evaluating the replacement algorithms. The hit ratio refers to the ratio of the number of requested files that can be found in cache to the number of total requested files. The byte hit ratio refers to the ratio of the volume of requested data that can be found in cache to the volume of total requested data. The hit ratio reflects the improvement on response time observed by clients, while byte hit ratio expresses the savings on the traffic to the remote storage facility.

We use trace driven simulations to compare hit ratio and byte hit ratio of our LVCT with those of LRU, GDS and LCB-K. It is suggested in [8] that K = 2 is sufficiently good and ensures an acceptable overhead

Access Times	1	2	3	4	> 4
Number of Files	115,930	34,514	17,772	10,770	28,345

Table 1. Breakdown of 207,331 accessed files according to their number of references.

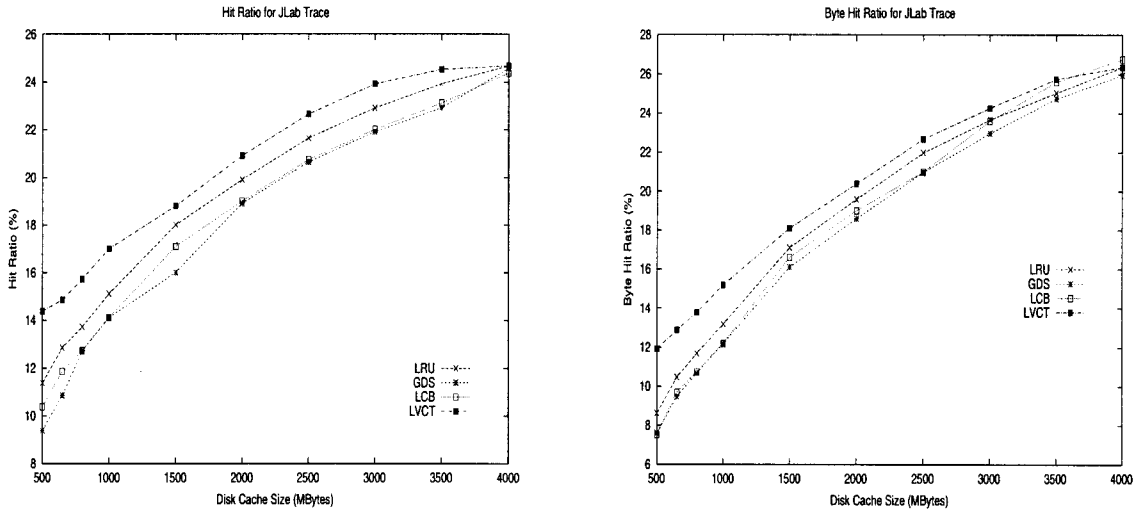

Figure 2. The hit ratio curves and byte hit ratio curves of JLab trace with various disk cache sizes for replacement policies LRU, GDS, and LVCT.

for LCB-K. So we use LCB-2 for the comparison. The trace we use is a real system workload trace from Jefferson's National Accelerator Facility (JLab), which reflects representative data grid access activities for scientific computing. It records the file access activities for a period of about 6 months at the mass storage system, JasMINE. The log describes the request submission time, the time when the request file starts to be retrieved from its source, the time when the file was delivered to the cache disk, and the file size in detail. There are 207,331 files accessed in the traces, and their total size is 144.9 TeraBytes. In our simulator we set the size of disk cache to range from 500 Gbytes to 4 Terabytes. Figure 2 show the hit ratios and byte hit ratios of replacement algorithms for LRU, GDS, LCB, and LVCT. All the ratios are for the whole 6-month traces. Both figures show the LVCT outperforms LRU, GDS and LCB.

To understand the reasons why LVCT performs better than the other polices, we show the breakdown of the 207,331 accessed files according to their number of requests within the 6-month period in Table 1. It is noted that 56% of accessed files are referenced only once in the period of 6-months. This is a distinctive characteristic of data access patterns in Grid computing different from those observed in Web-caching. Though Web requests also show strong popularity on a subset of objects, it is rare to have a large number of one-time-accessed objects for a long period of time [3]. Thus it is critical to test the eligibility of missed files before admitting them into cache in disk cache management. LRU, GDS, and LCB replacement policies admit any retrieving files into disk cache. Our simulations show that the average time for a admitted one-time-accessed file to stay in the cache without being referenced before being evicted is 1.45 days, 1.36 days and 1.31 days for LCB, GDS, and LRU respectively when the disk cache size is 1 Terabytes. Considering the large number of one-time-accessed files, their consumption of cache space is a significant waste of limited disk cache resources. Because these one-time-accessed files have infinite caching times, they are not admitted into disk cache at all in our LVCT replacement policy. This is also true for the files that have been evicted from cache for a long period of time and have large caching times. It is noted that these files are recorded in the caching time stack even though they are not admitted in the cache. By doing so, the files will have small caching times and pass the eligibility test to be admitted into cache if they are re-accessed soon. The

performance advantage of LVCT over the other algorithms at small cache size is especially pronounced, because disk caches of small sizes are more susceptible to the space waste caused by unjustified file caching.

6. Conclusion

A data replacement algorithm in SRM plays a crucial role in data access management for data grids. Identifying a critical weakness in existing replacement policies, we have presented a corrected utility function, and proposed an improved disk caching algorithm for data grid management. Our trace-driven simulation results show its consistent effectiveness compared with several representative replacement policies. We are making an collaborative effort to implement our replacement policy in data grids.

Acknowledgments: We thank Dr. Ekow Otoo from Lawrence Berkeley National Laboratory for providing us with data grid traces, and technical discussions.

References

[1] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin, "Evaluating Content Management Techniques for Web Proxy Caches", *Proceedings of the 2nd Workshop on Internet Server Performance*, May, 1999

[2] M. Abrams, C. Standbridge, G. Abdulla, S. Williams and E. Fox, "Caching Proxies: Limitations and Potentials", *Proceedings of 4th International World Wide Web Conference*, December 1995.

[3] L. Breslau, P. Cao, L. Fan, G. Phillips and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications", *Proceedings of Infocom'99*, April 1999.

[4] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms", *Proceedings of USENIX Symposium on Internet Technologies and Systems*, 1997

[5] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger, "Data Management in an International Data Grid Project", *Proceedings of 2000 IEEE/ACM International Workshop on Grid Computing*, Dec 2000

[6] S. Jiang and X. Zhang, "LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance", *In Proceedings of ACM SIGMETRICS 2002*, June 2002.

[7] P. Lorensetti, L. Rizzo and L. Vicisano, "Replacement Policies for a Proxy Cache", http://www.iet.unipi.it/luigi/research.html

[8] E. Otoo, F. Olken, and A. Shoshani, "Disk Cache Replacement Algorithm for Storage Resource Managers in Data Grids", *Proceeding of IEEE Conference on Super-Computing*, 2002

[9] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: simple and effective adaptive page replacement", *Proceedings of 1999 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1999, pp. 122-133.

[10] A. Shoshani, A. Sim, J. Gu, "Storage Resource Managers: Middleware Components for Grid Storage", *Proceedings of 19th IEEE Symposium on Mass Storage Systems*, 2002.

[11] R. Wooster and M. Abrams, "Proxy caching that estimates page load delays", *Proceedings of 6th International World Wide Web Conference*, April 1997.

[12] S. Williams, M. Abrams, C. Stanbridge, G. Abdulla and E. Fox, "Removal Policies in Network Caches for World-Wide Web Documents", *Proceedings of the ACM Sigcomm 96*, August, 1996.

[13] N. Yong, "Online File Caching", *Proceedings of 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998