Disk Cache Replacement Algorithm for Storage Resource Managers in Data Grids

Ekow Otoo, Frank Olken and **Arie Shoshani**
Lawrence Berkeley National Laboratory
1 Cyclotron Road, MS: 50B-3238
University of California
Berkeley, CA 94720

Abstract

We address the problem of cache replacement policies for Storage Resource Managers (SRMs) that are used in Data Grids. An SRM has a disk storage of bounded capacity that retains some N objects. A replacement policy is applied to determine which object in the cache needs to be evicted when space is needed. We define a utility function for ranking the candidate objects for eviction and then describe an efficient algorithm for computing the replacement policy based on this function. This computation takes time $O(\log N)$. We compare our policy with traditional replacement policies such as Least Frequently Used (LFU), Least Recently Used (LRU), LRU-K, Greedy Dual Size (GDS), etc., using simulations of both synthetic and real workloads of file accesses to tertiary storage. Our simulations of replacement policies account for delays in cache space reservation, data transfer and processing. The results obtained show that our proposed method is the most cost effective cache replacement policy for Storage Resource Managers (SRM).

Keywords and Phrases: file caching; cache replacement algorithm; trace-driven simulation; data staging; storage resource management.

1 Introduction

A storage resource manager (SRM) [13], in the context of the data-grid infrastructure [4, 9], is essentially a middle-ware component that facilitates the sharing of data and storage resources. A key function of its services is the management of a large capacity disk cache that it maintains. We address the problem of cache replacement policies for Storage Resource Managers (SRMs) used in data-grids. An SRM, described subsequently, maintains a large capacity disk for staging files and objects of varying sizes that are read from or written to Mass Storage Systems (MSS) that are either at the same local site or some remote site. Its role in the data-grid is analogous to that of a proxy server or a reverse proxy server in the World Wide Web. Although SRMs differ in many respects from proxy and reverse proxy servers, they share some common service functionalities such as caching of files or objects. One difference between caching in an SRM and caching in a web-server is that SRMs typically deal with batched requests of files or objects that are very large and incur significantly long delays in transferring and processing them. We

0-7695-1524-X/02 $17.00 (c) 2002 IEEE

address file or object replacement policies in SRMs, taking into account the latency delays in retrieving, transferring and processing of the files.

The *Grid* [6] may be described as a network of geographically distributed platforms of high performance heterogeneous computational nodes and data storage resources. Computational platforms range from super-computers and large scale cluster-computing farms to desktop workstations. Storage resources range from mass storage systems, e.g., tertiary storage system, high performance storage systems (HPSS), RAID farms and network attached storage(NAS) to workstation group servers. A *data-grid* is a network of geographically dispersed nodes of data and storage resource that is used to efficiently support data intensive applications through middleware services. A storage resource is accessible by a user, either remotely or locally for creating, destroying, reading, writing and manipulating *files.*

The dataset that is generated from some scientific experiments or observations say, is maintained in multiple storage resources that are distributed over wide area networks. We use the term dataset to mean a collection of related files that are specific to a defined activity of interest in a project. A file contains a collection of records each of which specifies a subset or the entire set of attributes that characterize an entity. The scientific experiments of particular project that is carried out over a number of years, result in the generation of datasets in the order of hundreds of terabytes to a few petabytes. These datasets are typically maintained in mass storage systems or tertiary storage systems. Client applications that run on either workstations or workstation clusters and super-computers request subsets of the dataset (i.e. a subset of the collection of files) that reside on direct attached storage (DAS), a tape storage system or even mass storage system at a remote site. We use the term *mass storage system or MSS* for any large capacity storage system, except DAS, that incur very long latency in retrieving the files resident on it.

Two significant decisions govern the operation of an SRM. Unlike web proxy severs, each of the requests that arrive at an SRM can be for hundreds or thousands of objects at the same time. As a result, an SRM generally queues the requests and subsequently makes decisions as to which files are to be retrieved into its disk cache. Such decisions are governed by a policy termed the *"file admission policy."* When a decision is made to cache a file it determines which of the files currently in the cache may have to be evicted to create space for the incoming one. The latter decision is generally referred to as a *"cache replacement policy"* and it is the subject of this paper in the context of storage resource managers in the data-grid. A replacement policy involves computing some utility function $\phi_i(t)$ for each of the files i, that potentially can be replaced and then replacing the one that has the minimum or the maximum utility function according to the criterion for replacement. Only files in the cache that are not being processed are candidates for eviction. Such files are said to be *"unpinned."* File that are in cache but are in use are said to be *"pinned."*

The performance measures of cache replacement policies are typically expressed by two metrics: the ***hit ratio*** and the ***byte hit ratio***. These metrics are briefly defined in sub-section 4.4. We introduce a third measure of goodness which we term the *average cost per reference.* An effective *replacement policy* maximizes either the hit ratio or the byte-hit ratio but minimizes the average cost per cache reference. Ideally, one strives for an *"optimal"* replacement policy for a particular metric measure. The quest for optimal replacement policies is a long standing problem.

We make three main contributions to disk cache replacement policies in this paper. First we introduce a definition of a utility function and describe briefly an efficient algorithm for evaluating it during cache replacements. The minimum utility value for a set of N unpinned files is computed in time $O(logN)$. The utility function $\phi_i(t)$ is based on the solution of the fractional knapsack problem under the assumption that the cache capacity is sufficiently large and holds a significantly large number of files. The utility

function is expressed as:

$$\phi_i(t) = \frac{k_i(t)}{(t - t_{-k_i})} * \frac{g_i(t) * c_i(t)}{s_i};$$

where, for each file i, s_i is its size, $k_i(t)$ 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 and $c_i(t)$ is the cost of retrieving the file from its source into the cache at the time t. We call the cache replacement policy using the above utility function, the *least cost beneficial based on the K backward references* or *LCB-K policy* for short. Second, we define a new measure for cache replacement policies which we term *"average cost per reference"*. Using both synthetic and real workload, we show that a policy that replaces the file with the minimum utility function as defined above gives the minimum average cost per reference compared with other known replacement policies such as *random (RND)*, *least frequently used (LFU)*, *least recently used (LRU)*, *least recently used based on K backward references (LRU-K)* and *Greedy Dual Size (GDS).* The LCB-K policy does not necessarily maximize either the hit ratio or the byte hit ratio. Third, we briefly present the correct approach to simulating a disk cache replacement policies in wide area networks, taking into consideration delays incurred in retrieving the file from its origin, transferring it into the cache or staging disk and holding it in cache for processing.

2 Configuration and Related Works

The use of an SRM helps to ameliorate the latency experienced when users request files and objects with long retrieval times. Its role is to retain in its disk cache, a single copy of a file that is requested and then use it to service subsequent requests for the same file. Its usage is similar to that of a *proxy server* or a *reverse proxy server*, except that SRMs deal with transfers of objects that are of the order of gigabytes in size. Figure 1 depicts the positional role of an SRM within a data-grid. A storage resource manager may be specialized to be either a *disk resource manager (DRM)* or a *hierarchical resource manager (HRM)*. A DRM services requests directed to it from either its disk cache or by first retrieving the file from another remote SRM into its cache. An HRM acts as a front end to a tertiary storage system. It services requests either from its disk cache or retrieves the file from a tertiary storage that is directly connected to it, into its disk cache.

In environments that deal with file transfers from archival tape over wide area networks, the strategy of caching files in disk storage at a site along the routing path and then forwarding it to the client has been called *Data Staging* [14, 15]. This significantly improves the data access response times. Another related area where the issues we address in this paper have applicability is in *Web-Caching* [1, 3, 5, 7]. Caching techniques are effective where file reference streams exhibit:

- *Locality of Reference:* A file that is referenced and read into a cache is referenced multiple times by the same user over a very short period.
- *Shared Access to Files:* The same file after it is read into cache, is also referenced by different users.

Some earlier works on file caching in distributed systems and the staging of files from tertiary storage have been presented in [8, 10, 14, 15, 16, 18]. Recent studies on caching have focused more on web-caching [1, 3, 5, 12, 17]. Cao and Irani [3] present a relative comparison of various cache replacement policies that have been proposed for web-caching. Their work discusses some of the merits and concerns

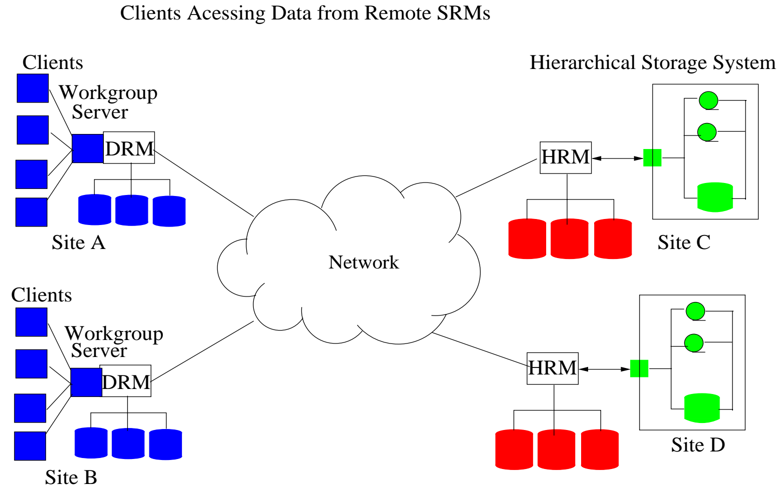


Figure 1: Use of a Storage Resource Manager in a Data-Grid

of replacements policies such as Least Frequently Used (LFU), Least Recently Used (LRU), etc. They propose a replacement policy for web-caching called the *Greedy-Dual-Size (GDS)* [3]. It is a variant of the replacement policy termed *Greedy-Dual (GD)* [18], that was originally proposed for main memory caching of fixed size pages. It is also perceived as a generalization of the LRU when there is some variability in the cost of reading pages from secondary storage. We have included an implementation of the GDS policy with file transfer and processing delay taken into account.

2.1 Differences Between Caching in SRMs and Web-Caching

In principle the files or objects in web-caching can be of any media type and of varying sizes. However web-caching in proxy servers are realistic only for documents, images, video clips and objects of moderate size in the order of a few megabytes. On the other hand, the files in SRMs have sizes of the order of hundreds of megabytes to a few gigabytes. Some of the differences between caching in SRMs and web-caching are summarized in Table 1.

3 Our Utility Function for Ranking Files for Eviction

The basic idea of our page replacement policy is to evaluate the utility function $\phi_i(t)$ for each file i in the disk cache. An object i of size s_i has a retrieval cost $c_{i,r}(t)$ from site r. The retrieval costs vary with time depending on when and where the file is fetched. In the environment of the data-grid there could be replicas of the same file at different sites r. We will denote the cost simply by $c_i(t)$, with the understanding that this is the minimum cost over all replica sites. At each instant in time t, when we need to acquire space of size s_j for a file j, we order all the unpinned files in non-decreasing order of their utility functions and evict the first m files with the lowest values of $\phi_i(t)$ and whose sizes sum up to or just exceed s_j. We always assume that the sizes of the cached files are relatively small compared to the total size S, of the cache.

Characteristic Property	Web Caching	Disk Caching in SRMs
File/Object Size	Variable size objects of the order of megabytes	Variable size objects of the order of gigabytes
Cache Size	In the order of tens to hundreds of gigabyte	In the order of hundreds of gigabyte to tens of terabytes
Source Latency	A few milliseconds to seconds	In milliseconds to minutes
Object Transfer Time	In milliseconds to a few minutes	In seconds up to a few hours
Duration of Object Reference	Almost Instantaneous	In seconds up to a few minutes
Caching Requirement	Optional	Mandatory
Batched Requests	Typically one request references one object but may have a additional references to linked objects.	May involve thousands of files in one request.
Bundle Constraint	Only one object is referenced per request.	May require that multiple files be accessed simultaneously.
Cache Consistency	Cognizant of modified documents	Predominantly Read-Only and ignores consideration of cache coherence
Network bandwidth requirement	Standard Internet	High speed gigabit networks

Table 1: Summary of Differences between Caching in SRMs and Web-Caching

To see why this holds consider a request stream to an SRM. Let this stream be denoted by a sequence $r_1, r_2, r_3, \ldots, r_i \ldots$, of random variables with a common stationary probability distribution $p_1(t), p_2(t)$, $p_3(t), \ldots, p_n(t)$, where $p_j(t)$ is the probability that r_i references the file j, i.e., $Prob(r_i = j) = p_j(t)$ for all $i \geq 1$ and n is the entire number of files to be accessed. Our objective is to minimize the total cost of retrieving the files that occupy the cache. Let the set of files that are retrieved into the cache be C, where $C \subseteq I = \{1, 2, \ldots, n\}$. The collection of files that are accessed are identified by the integer numbers in I. Then an optimal cache replacement always retains the set C in the cache such that

$$\sum_{j \in (I-C)} p_j(t) * c_j(t); \qquad (1)$$

is minimized subject to

$$\sum_{i \in C} s_i \leq S. \qquad (2)$$

Since we assume that the sizes of the cached objects are relatively small compared to the total size S of the cache, the amount of space left after the maximum number of files are cached is negligible. Since we restrict the solution to the set C satisfying $\sum_{i \in C} s_i = S$, we can restate the problem as

$$maximize \sum_{i \in C} p_i(t) * c_i(t); \qquad (3)$$

subject to

$$\sum_{i \in C} s_i = S. \tag{4}$$

Under the above assumption then, the problem statement expressed by 3 and 4 is equivalent to that of the *Knapsack* problem. This is well known to be NP-hard for which there is no known optimal algorithm. On the other hand a corresponding *fractional knapsack* problem has an optimal solution. An optimal solution for the *fractional knapsack* is given by a simple greedy algorithm as follows. One ranks the files $i \in I$ in non-increasing order of the ratios of the contributing cost to size, i.e., $p_i(t) * c_i(t)/s_i$ and then retain in the cache the items beginning from the first to the last until either all items are retrieved into the cache or the constraint (2) is just satisfied.

3.1 Cost Beneficial Cache Replacement Policy

The cache replacement policy we propose, is based on the utility function defined in the preceding subsection. Restated differently, we have that whenever a file in the cache needs to be evicted at time t, the eviction candidate is the one that has the minimal utility function $\phi_i(t)$ given by

$$\phi_i(t) = \frac{p_i(t) * c_i(t)}{s_i} \tag{5}$$

Similar conclusions have been reached in [10, 12] but under different assumptions. The problems studied for which they derived their results were different. The utility function as expressed by equation (5), is not practical to apply since we do not know the probabilities and even more, these probabilities are not stationary. Under the assumption that the references to the objects are independent, the distribution of the inter-reference times of each individual file i can be approximated by an exponential distribution. Alternatively, the number of reference made to each individual file i can taken to be a Poisson distribution with parameter λ_i. The probability term in equation (5) can be replaced by the approximation

$$p_i(t) = \frac{\lambda_i(t)}{\sum_{1 \le j \le n} \lambda_j(t)}.$$

Since the replacement decision is based only on the relative rankings of $\phi_i(t)$ our utility function of equation of 5 may now be written as

$$\phi_i(t) = \lambda_i(t) * \frac{c_i(t)}{s_i}. \tag{6}$$

To estimate the values of $\lambda_i(t)$ we utilize the idea applied in developing the *Least Recently Used Based on the K backward reference (or LRU-K),* page replacement policy. We modify the idea in the *LRU-K* [11] algorithm slightly by retaining the times of the last $k_i(t)$ up to a maximum of K. Let the time of the k_i^{th} backward reference be denoted by t_{-k_i}. Then we can estimate the rate of arrival by

$$\lambda_i(t) = \frac{k_i(t)}{t - t_{-k_i}}$$

The cost of the future retrievals is also not known. We utilize a best effort estimate, denoted by $\hat{c}_i(t)$, by

deriving it from the last $k_i(t)$ retrieval made. Note that before a file becomes a candidate for eviction, at least one retrieval of the file must have been made at some time in the past. The rate of arrival is computed from the $k_i(t)$ most recent references. However the file could have been referenced multiple times in the past and this fact needs to be considered in evaluating whether the file should be retained in cache or not. We account for this by factoring in the cumulative count of accesses to the file. Let this be denoted by $g_i(t)$ for file i. Our eviction candidate then becomes that file with the minimum value of $\phi'_i(t)$ where

$$\phi'_i(t) = \frac{k_i(t)}{t - t_{-K_i}} * \frac{g_i(t) * c'_i(t)}{s_i} \tag{7}$$

The above expression requires that we maintain the counts $g_i(t)$, for all files that are in cache and out of cache. This is not practical and therefore we maintain the reference counts for only the files that have been accessed since some time T in the past. The value of T defines the lifetime of an active file. A file becomes active when it is first referenced. The history of the accesses to it is held in memory even after the file is removed from cache. However, if the file is never referenced again after the lifetime T, it is purged from memory and the information on the history of accesses is lost. Any subsequent access to the file would begin accumulating a new history.

Our cache replacement policy, based on the equation (7) is referred to as a *Least Cost Beneficial* replacement policy based on at most K backward references or *LCB-K* for short. Using equation (7), the problem of implementing an efficient algorithm for ranking the files in cache is still non-trivial. Note that the utility function given by equation 7, does not provide a *stationary ranking relation* [2]. Consequently a simple priority queue cannot be used to order the files in logarithmic time. However it gives a new insight into implementing a variant of the LRU-K algorithm. Since LRU-K assumes constant retrieval cost and fixed size pages, we can rank the files only by $(t - t_{-K_i})/k_i(t)$ and evict the file with the maximum value (or maximum inter arrival time computed from the K backward references) We call this policy the *MIT-K* replacement policy. A fast implementation makes use of at most K priority queues. The MIT-K replacement policy is implemented in our experiments instead of the original LRU-K.

Our strategy for quickly evaluating the utility function involves defining m intervals into which the computed value of $k_i(t) * g_i(t) * c'_i(t)/s_i$ can fall. The files that fall within each interval are ranked according to the utility function given by equation 7. The partitions or interval sizes are periodically updated.

4 The Cache Replacement Algorithm

4.1 On Correctly Simulating Disk Cache Replacement Policies

In simulating cache replacement polices in SRMs, or any disk caching environments such as Web-Caching, one needs to account specifically for three types of delays:

i the latency incurred at the source of the file;

ii the transfer delay in reading the file into the disk cache; and

iii the holding or pinning delay incurred while a user processes the file after it has been cached. This may simply involve transferring the file into the user's workspace.

Time	File Id / Size	f_1	f_2	f_3	f_4	f_5
	Size	2	3	2	4	1
1		r_1				
2			r_2			
3					r_3	
4		r_4				
5				r_5		
6					r_6	
7						r_7
8			r_8			

(a)

Time	File Id / Size	f_1	f_2	f_3	f_4	f_5
	Size	2	3	2	4	1
1		r_1				
2			r_2			
3						
4		r_4				
5					r_3	
6						
7					r_6	
8				r_5		
9						r_7
10			r_8			

(b)

Table 2: An Example of a Reference Stream of File Accesses

Simulations of cache replacement policies that have been studied for database buffering, tertiary storage file staging and web-caching do not normally take these into account. To appreciate the significance of these delays, consider the following file reference stream $\{r_1, r_2, \ldots, r_8\}$, that consists of accesses to files $f_1(2), f_2(3), f_3(2), f_4(4)$ and $f_5(1)$. The notation $f_X(Y)$ implies that file f_X has size Y megabytes. Let the cache size for these requests be 7 megabytes say. Table 2a shows the time line of request arrivals. For example at time t = 2, the request r_2 asked for file f_2 to be cached.

Suppose we take a simple first-come-first-serve (FIFO) scheduling policy for admitting the request to be serviced. If a request encounters its file in cache, it immediately begins processing it. A file that is being processed is pinned in the cache for the duration of time that it is processed. Let the sequence $D_{cache} = \{1, 1, 2, 2, 1, 2, 1, 1\}$, denote the duration of retrieval times that correspond to the respective requests and suppose the sequence $D_{proc} = \{3, 1, 2, 3, 2, 2, 1, 1\}$ specifies the durations of processing times of the respective requests. The simulation of an LRU policy, if we ignore the delays, generates the following sequence of evictions: r_3 causes f_1 to be evicted; r_4 causes f_2 to be evicted, r_5 causes f_4 to be evicted; r_6 causes f_1 to be evicted and so on. However if we take the retrieval and processing delay into consideration, we get the following eviction sequence: r_3 causes f_2 to be evicted and r_5 causes f_1 to be evicted. The simulation of the LRU policy, with no delays considered, does not encounter any cache hits. However the same LRU policy, when delays are accounted for, encounters two cache hits. LRU replacement policy with delay considerations under our simply FIFO admission policy, generates the schedule of Table 2b. We are unaware of any simulations of cache replacement policies that account for the delays in processing.

4.2 Data Structures for Cache Replacement Algorithms

All the cache replacement algorithms addressed in this paper make use of three fundamental data structures. A search tree T_s for holding the information of all accessed files, a container C_p that holds information of files that are pinned and another container C_u that holds files that are in cache but not pinned.

The algorithms differ only with respect to the data structure C_u, and the algorithm for evicting an unpinned file. We give some details on the implementation that are specific to the LCB-K policy. The data structures have equivalent container types from the Standard Template Library (STL).

Search Tree T_s: This a Red-Black tree whose nodes hold the identifiers of files that have been accessed. Further, the node corresponding to a file i also stores values for $t_i(q), k_i(j), g_i(j), c'_i(q), 1 \leq j \leq K, 1 \leq q \leq k_i$, Where $t_i(q)$ is a *" defined referenced time"* of the file, as well as a status indicator that specifies whether the object is pinned, is in cache but unpinned or has been evicted.

A Vector C_p: In particular an element in this container stores the count of the number of pins held on a file. Whenever a request completes processing a file it decrements the pin count by 1 and if this value becomes zero, the element is migrated to the container C_u described below.

A Vector C_u: This is a vector container of m priority queue like where m is the number of partitions of the values $k_i(t) * g_i(t) * c'_i/s_i$. The root node of $C_u[q]$ corresponds to the file with the minimum utility function value for all those unpinned files that fall in the interval q. The root nodes form a tournament of priority-queues. The file that corresponds to the root node whose utility function value is minimum is the actual candidate for replacement.

Given the supporting data structures in the implementation of the replacement algorithms, it is straight forward to see how we achieve the logarithmic time in selecting the eviction candidate in the LCB-K policy.

4.3 The Simulation Model for Cache Replacement Policies

The simulations of the replacement policies are done as discrete event simulations. Each request r_i provides five distinct event times. These are: *Arrival_Time(r_i)*, *Start_Caching(r_i)*, *End_Caching(r_i)*, *Start_Processing(r_i)* and *End_Processing(r_i)*. An event object *(r_i = evtObj)*, is created upon an arrival of a request. This is then inserted into an event queue denoted by *EvtQueue*. An event object *evtObj* has two fields, *eventType(evtObj)* and *schdTime(evtObj)*, that identify respectively the type of event and the scheduled time at which that event should occur. The *EvtQueue* is implemented as a priority queue.

The action taken upon the occurrence of an event is implied by its type. We give the semantic action of only one to illustrate the idea. Suppose an event object *evtObj*, is removed from the root of the event queue. If the event type given by *eventType(evtObj)* is a "Start_Caching" event, then the *eventType(evtObj)* is set to *"End_Caching"* and the *schdTime(evtObj)* is set to the scheduled completion time for retrieving the file into the cache. The event object *evtObj*, is then reinserted into the priority queue *EvtQueue*.

The simulation is driven by a workload of file requests. Suppose the time of arrival of a request r_i is t_0 and assuming the root node of a non-empty *EvtQueue* is denoted by *EvtQueue(Root)*. If $t_0 \geq$ *schdTime(EvtQueue(Root))* then *EvtQueue(Root)* is removed and assigned to *evtObj*. The simulation then executes the actions corresponding to the *eventType(evtObj)*. If $t_0 >$ *schdTime(EvtQueue(Root))* then a new event object is created using the information of the arriving request. This is then inserted into the *EvtQueue*.

We should also remark that of the five event times of a request, the simulation only checks if the requested file is in cache at the time when a *Start_Caching* event occurs. If the file is in cache the request is immediately schedule for processing. However if the file is not in cache then a cache replacement algorithm is executed on the vector container C_u to free enough space to retrieve the file. This is considered also as a space reservation phase.

4.4 Performance Metrics

The traditional performance metrics that have been used to measure the effectiveness of a cache replacement policy are the *hit ratio* and the *byte hit ratio*. Given a request stream (or workload), the hit ratio is defined as the ratio of the requests that find their files in cache (i.e., hit their files in cache), to the total number of requests. A byte hit ratio is the ratio of the volume of data (in bytes) that are encountered in the cache to the total volume of data requested.

In any case these measures provide some insight into the improvement in response times and savings in bandwidth utilization due to caching. Hit ratio gives the relative savings as a count of the number of files hit, while the byte hit ratio measures the relative savings in the volume of data prevented from being transferred. The measure of byte hit ratio gives some indication in the savings in bandwidth usage and consequently improvement in the response times of client requests. None of these measures accounts for the latency incurred either at the data source or in processing the file after it is cached. For example when the data source is from a robotic tape device where the delay can sometimes be comparable to the data transfer time, the measure of byte hit ratio does not reflect such delays. We introduce a third measure which we call the *"Average Cost Per Reference"*. This is defined as the ratio of the total cost (in time units) of retrievals to the total number of references. This gives a better indication of the relative savings in time to retrieve and transfer a file into the cache.

5 Performance Comparison of Some Replacement Policies

We compared the performance metrics of *hit ratio*, *byte hit ratio* and *average cost per reference* for a number of cache replacement policies, namely RND, LFU, LRU, MIT-K (i.e., a variant of LRU-K), GDS and LCB-K. We set the active lifetime T to be five days. That is if a file has not be accessed in the last five days, it is purged from memory. The simulations were conducted for both synthetic and a real workload of a mass storage system. The real system workload is a log of file access activities, for about a six months period, of the mass storage system, JasMINE, at the Jefferson's National Accelerator Facility (JLab). This is the only mass storage system logs that we were able to find that contains the required information for our simulations. The file caching being done in JasMINE uses the LRU policy. We expect the LRU, MITK and GDS policies to give better performance measures for this workload.

The sizes of files ranged from about 1.0 to 2.1 gigabytes and the time scales are in the order of minutes. Arrivals are batched, in the sense that for the same request, the arrival time can be associated with multiple files. Using the time scale and file sizes of the workload from JLab as a guide, we generated a synthetic workload based on a Poisson inter-arrival time with mean 90 seconds. The file sizes, in bytes, are uniformly distributed between 500,000 and 2,147,000,000. The entire period of request generation is broken into random intervals and we inject locality of reference using the 80-20 rule. That means that within each interval of the request stream, 80% of the requests are directed to 20% of the files. The length of an interval is uniformly distributed between 1% and 5% of the generated workload.

5.1 Experimental Results

Figure 2 shows the graphs of the performance metrics of the workload from JLab and Figure 3 shows the same plots for the synthetic workload. The plots are generated from runs using variance reduction technique. Each workload has just over 500,000 entries. For each specified value of the cache size and for each workload, a run generates 5 values of the required performance metric, at intervals of about

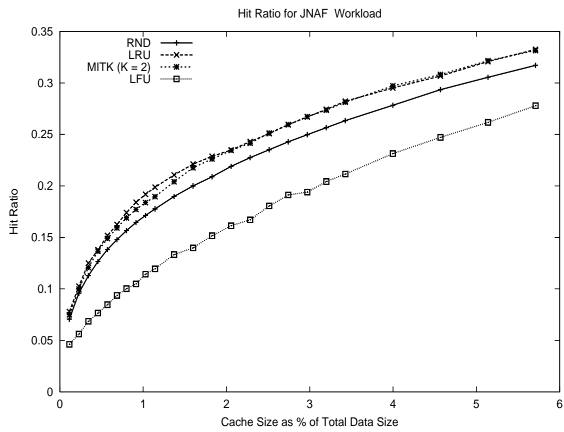
100,000 requests. Each point in the graphs, for any particular measure, is the average of the 5 recorded measures in a run.

The graphs for each workload are shown in two columns. The graphs in the first column, show the relative performance measures for the RND, LFU, LRU and MIT-K for K = 2. These policies ignore the retrieval costs and file sizes in computing the utility function. In these plots, MIT-K for K = 2, consistently gave good performance measures. For K = 2, the performance practically the same as that of LRU.

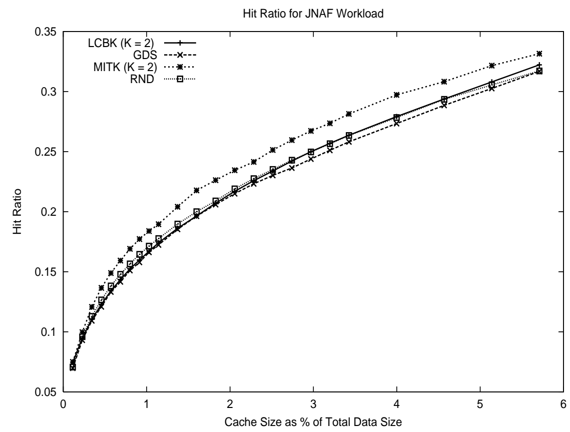
The graphs in the second column show the relative performance measures for RND, MIT-K, GDS and LCB-K (K = 2). GDS and LCB-K make use of the retrieval costs and file sizes in computing the utility functions. RND is included in this set of graphs because we use this as the base line graph for comparisons. Since MIT-K gives the best performance measures in the policies that ignore both retrieval costs and file sizes, we include this also in the set of graphs in the second column to show how it compares with GDS and LCB-K.

Figures 2a and 2b show the graphs of the hit ratios while figures 2c and 2d show the graphs of the byte hit ratios. Although the MIT-K and LRU give the best results, their graphs are not significantly different from those of GDS and LCB-K. On the other hand, examination of the graphs of the average retrieval cost per reference, shown in figures 2e and 2f, indicate that LCB-K and GDS give best performance measure. Under all performance measures LFU gives the worst performance.

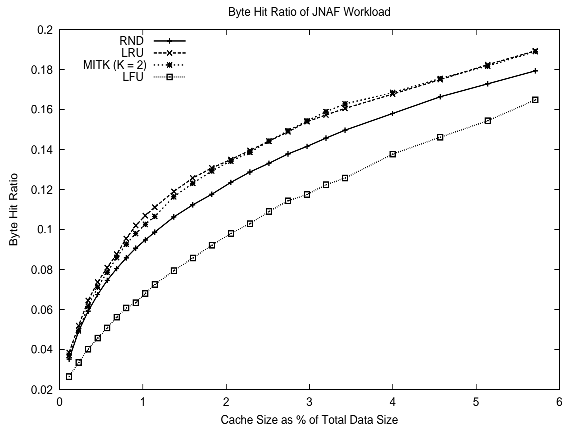
The corresponding graphs for the synthetic workload shown in figure 3, illustrate the same relative performance measures of the various policies as indicated by a real workload. In the development of LRU-K, the authors suggested that values of K = 2 or 3 is sufficient and recommended the use of K = 2. We confirmed this fact in our simulations. Due to lack of space we leave out the supporting graphs.



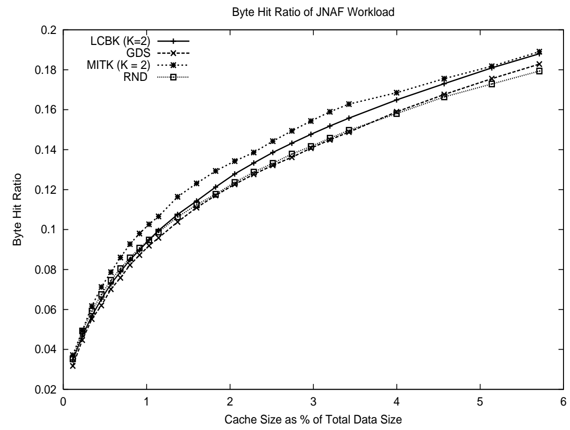
(a)



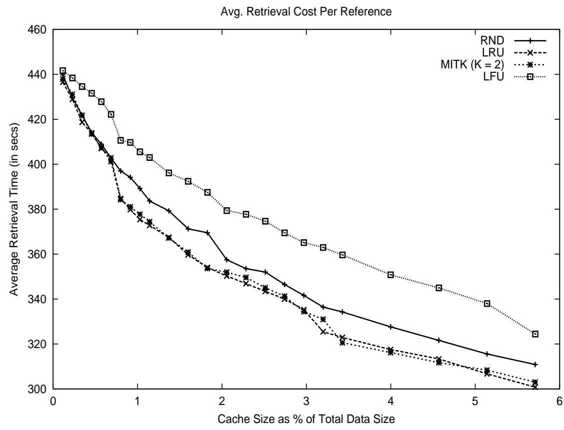
(b)



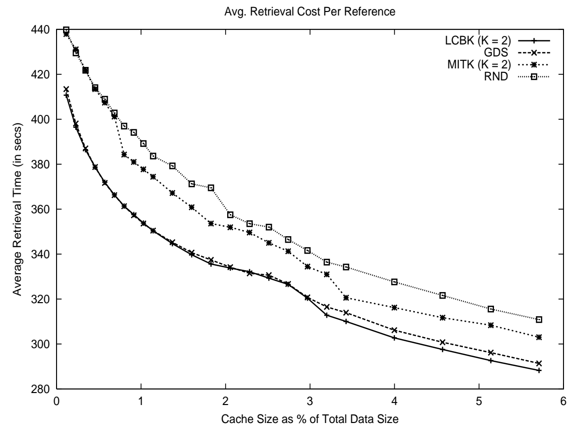
(c)



(d)

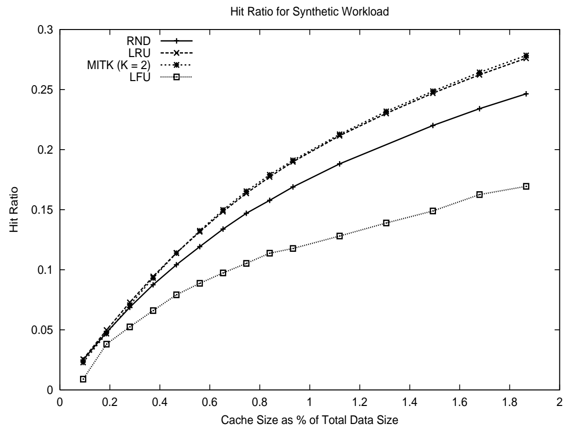


(e)

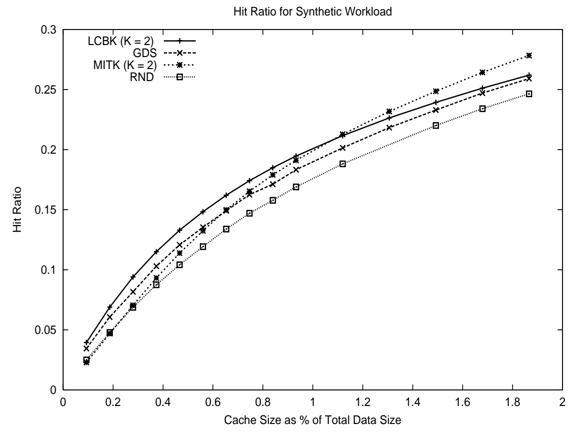


(f)

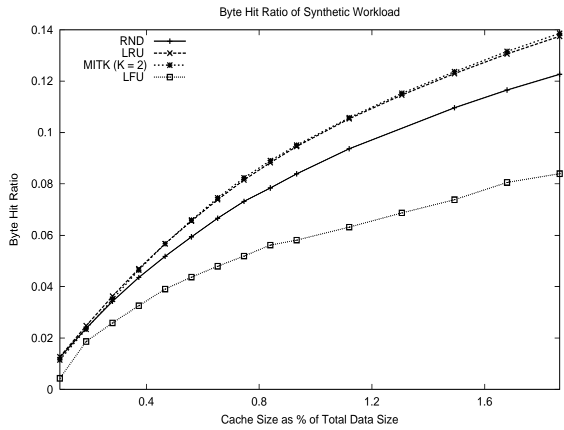
Figure 2: Graphs of for Real Workload from Jefferson's National Accelerator Facility (JLab)



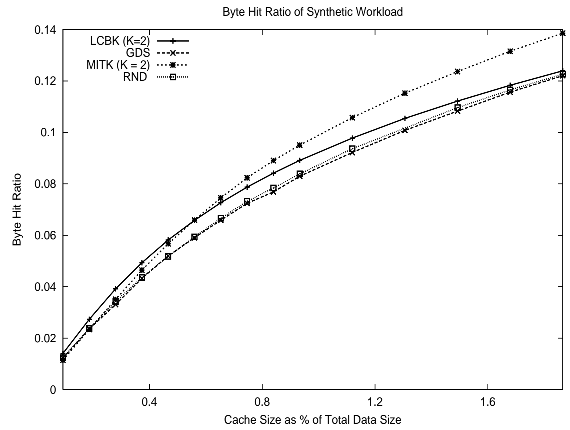
(a)



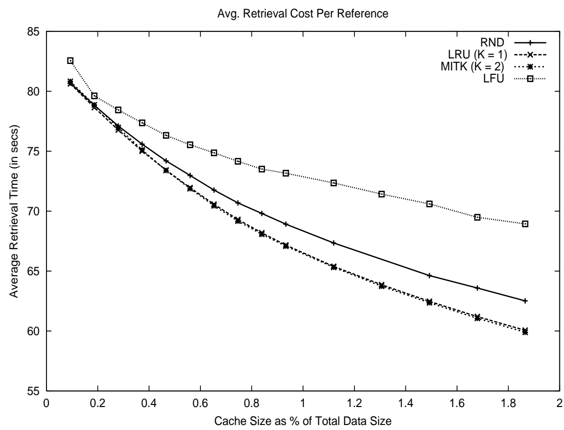
(b)



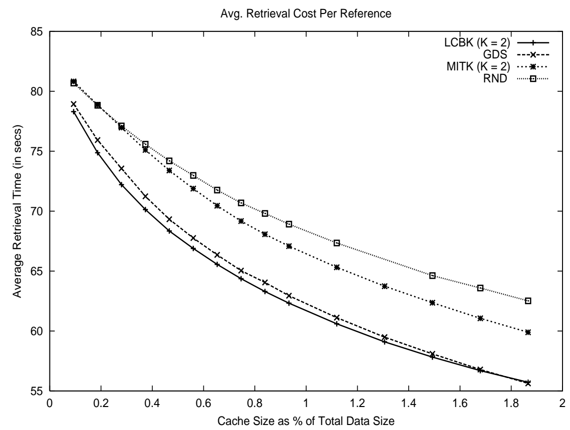
(c)



(d)



(e)



(f)

Figure 3: Graphs of Performance Metrics for Synthetic Workload

6 Conclusion and Future Work

Disk file caching in storage resource managers, has some characteristic features that distinguish it from caching in other domains such as virtual memory, database page buffering and web-caching. In particular, disk file caching in SRMs involve variable size files or objects that are very large and the delays caused by source latency, file transfers and processing of the files significantly affect its performance.

We have defined a utility function for determining which files need to be evicted from the cache of an SRM when space is needed. We have presented an efficient method for evaluating the replacement policy based on the utility function using a tournament of priority queues. Unlike traditional simulations of cache replacement policies we also presented a realistic simulation model that accounts for the delays in processing objects in the cache. Using the simulation model, we compared the replacement policies of RND, LFU, LRU, GDS, MIT-K and LCB-K under the performance metrics of hit ratio, byte hit ratio and average cost per reference for both synthetic and actual workloads. We conclude that average cost per reference is the most realistic performance metrics for evaluating disk cache replacement policies for storage resource managers and under this performance measure the least cost beneficial replacement policy gives the best result of the policies compared.

Future work in this area would involve more extensive testing with real workloads from other mass storage systems at Fermi Laboratory and the high the performance storage system (HPSS) of the National Energy Research Computing (NERSC) center. The LCBK and the GDS are being planned for inclusion in storage resource managers deployed in data grids.

Acknowledgment

We would like to express our sincere gratitude to Andy Kowalski of Jefferson's National Accelerator Facility and Don Petravick of Fermi National Laboratory for providing us with the workloads used in our simulation runs. This work is supported by the Director, Office of Laboratory Policy and Infrastructure Management of the U. S. Department of Energy under Contract No. DE-AC03-76SF00098. This research used resources of the National Energy Research Scientific Computing (NERSC), which is supported by the Office of Science of the U.S. Department of Energy.

References

[1] C. C. Aggarwal and P. S. Yu. On disk caching of web objects in proxy servers. In *Proc. Int'l. Conf. Info and Knowledge Management, CIKM'97*, pages 238 – 245, Las Vegas, Nevada, 1997.

[2] A. V. Aho, P. J. Denning, and J. J. Ullman. Principles of optimal page replacement. *J. ACM*, 18:80 – 93, Jan. 1971.

[3] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[4] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *J. Network and Computer Applications*, 23(3):187 – 200, 2000.

[5] S. Dar, M. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proc. VLDB Conf.*, pages 330 – 341, Bombay, India, Sept. 1996.

[6] I. Foster and C. Kesselman, editors. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publ., San Fracisco, 1999.

[7] D. Guerrero. Caching the web, part 1. *Linux Journal*, 57, Jan. 1999.

[8] U. Hahn, W. Dilling, and D. Kaletta. Adaptive replacement algorithm for disk caches in hsm systems. In *16 Int'l. Symp on Mass Storage Syst.*, pages 128 – 140, San Diego, California, Mar. 15-18 1999.

[9] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger. Data management in an international data grid project. In *Proc. 1st IEEE/ACM Int'l. Workshop on Grid Computing*, India, 2000.

[10] F. A Olken. HOPT: A myopic version of the STOCHOPT automatic file migration. In *Proc. ACM SIGMETRIC Conf. on Mesurement and Modelling of Comput. Syst.*, pages 39 – 43, Minneapolis, Minnesota, Aug. 1983.

[11] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database buffering. In *Proc. ACM SIGMOD'93: Int'l. Conf. on Mgmnt. of Data*, pages 297 – 306, Washington, DC, May. 1993.

[12] P. Scheuermann, J. Shim, and R. Vingralek. Watchman: A data warehouse intelligent cache manager. In *Proc. 22nd VLDB Conference*, pages 51 – 62, Bombay, India, Sept. 1996.

[13] A. Shoshani, A. Sim, and J. Gu. Storage resource managers: Middleware components for grid storage, Apr. 15 - 18 2002.

[14] A. J. Smith. Analysis of long term file reference patterns for application to file migration algorithms. *IEEE Trans. on Soft. Eng.*, SE-7(4):403–417, Jul. 1981.

[15] M. Tan, M.D. Theys, H.J. Siegel, N.B. Beck, and M. Jurczyk. A mathematical model, heuristic, and simulation study for a basic data staging problem in a heterogeneous networking environment. In *Proc. of the 7th Hetero. Comput. Workshop*, pages 115–129, Orlando, Florida, Mar. 1998.

[16] T. Theodore Johnson and E. L. Miller. Performance measurements of tertiary storage devices. In A. Gupta, O. Shmueli, and J. Widom, editors, *VLDB'98, Proc. 24th Int'l. Conf. on Very Large Data Bases*, pages 50 – 61. Morgan Kaufmann, Aug. 24-27 1998.

[17] S. Williams, M Abrams, C. Stanbridge, G. Abdulla, and E. Fox. Removal policies in network caches for world-wide-web documents. In *Proc. of ACM SigComm Conf.*, 1999.

[18] N. Young. On-line file caching. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1998.