Efficient Algorithms for Multi-file Caching

Ekow J. Otoo[1], Doron Rotem[1], and Sridhar Seshadri[2]

[1] Lawrence Berkeley National Laboratory, 1 Cyclotron Road
University of California, Berkeley, California 94720
[2] Leonard N. Stern School of Business, New York University
44 W. 4th St., 7-60 New York, 10012-1126

Abstract. Multi-File Caching issues arise in applications where a set of jobs are processed and each job requests one or more input files. A given job can only be started if all its input files are preloaded into a disk cache. Examples of applications where Multi-File caching may be required are scientific data mining, bit-sliced indexes, and analysis of sets of vertically partitioned files. The difference between this type of caching and traditional file caching systems is that in this environment, caching and replacement decisions are made based on "combinations of files (file bundles)," rather than single files. In this work we propose new algorithms for Multi-File caching and analyze their performance. Extensive simulations are presented to establish the effectiveness of the Multi-File caching algorithm in terms of job response time and job queue length.

1 Introduction

Caching techniques are widely used to improve the performance of computing systems whenever computations require frequent transfers of data between storage hierarchies that have different access speeds and/or network bandwidth characteristics. Given a sequence of requests for files from some slow or remote storage media, one may use a cache of a small relative size on a faster storage media that holds the most frequently used files. Retrieval to the slow or distant memory is needed only in the case that the file requested is not found in the cache. This results in improved efficiency and reduced costs even with relatively small cache sizes. When a requested file is not found in the cache the system incurs a "fault". The costs associated with a "fault" consist of costs of resources needed to read the file from slow memory and then transferring the file across the network. Efficient caching algorithms choose which current files in the cache must be replaced with new ones in order to maintain a set of files in the cache that minimizes the expected cost of "faults".

There are many papers [1,2,3,4,5,6] describing and analyzing caching and replacement policies. These works distinguish between online and off-line algorithms. In both cases, a sequence of requests for files arrive at a queue and must be served on a First Come First Served (FCFS) basis. A replacement decision must be made whenever a "fault" occurs. Online algorithms make a replacement decision based only on the current request and previous requests but do

F. Galindo et al. (Eds.): DEXA 2004, LNCS 3180, pp. 707–719, 2004.

not have any information about future requests. On the other hand, off-line algorithms make replacement decisions based on the complete sequence of both past and future requests. Off-line algorithms are not practical and are mainly used for establishing bounds and gaining insights on the performance of online algorithms.

In addition, caching algorithms can be classified based on their sensitivity to file sizes and "fault" costs. The following cases are considered in the literature:

Paging: Sizes of all files and their "fault" costs are equal
Fault Model: File sizes are arbitrary while "fault" costs are the same for all files
Weighted caching: All files sizes are the same but "fault" costs may be arbitrary
Bit Model: Files may have arbitrary sizes, "fault" costs are proportional to file size
General Model: Both "fault" costs and file sizes may be arbitrary

This work is motivated by file caching problems arising in scientific and other data management applications that involve multi-dimensional data [3,7,8]. The caching environment for such applications is different than the works described above in two main aspects:

Number of files associated with a request: As explained below due to the nature of the applications a request may need multiple files simultaneously. A request cannot be serviced until all the files it needs are in the cache.
Order of request service: In case several requests are waiting in the queue, they may be served in any order and not necessarily in First Come First Serve order (FCFS). Policies that determine the order in which requests are served (admission policies), become important and sometimes must be considered in combination with cache replacement policies [3].

1.1 Motivating Examples of Applications

Scientific applications typically deal with objects that have multiple attributes and often use vertical partitioning to store attribute values in separate files. For example, a simulation program in climate modeling may produce multiple time steps where each time step may have many attributes such as temperature, humidity, three components of wind velocity etc. For each attribute, its values across all time steps are stored in a separate file. Subsequent analysis and visualization of this data requires matching, merging and correlating of attribute values from multiple files. Another example of simultaneous retrieval of multiple files comes from the area of bit-sliced indices for querying high dimensional data [8]. In this case, a collection of N objects (such as physics events) each having multiple attributes is represented using bit maps in the following way. The range of values of each attribute is divided into sub-ranges. A bitmap is constructed for each sub-range with a $'0'$ or $'1'$ bit indicating whether an attribute value is in the required sub-range. The bitmaps (each consisting of N bits before compression) are stored in multiple files, one file for each sub-range of an attribute. Range

queries are then answered by performing Boolean operations among these files. Again, in this case all files containing bit slices relevant to the query must be read simultaneously to answer the query.

1.2 Problem Description

Our approach for caching multiple files consists of two steps that are applied at each decision point of the algorithm. Given a cache of some fixed size and a collection of requests currently waiting in the admission queue for service:

Step-1, File removal: We first remove from the cache a set of "irrelevant" files. Files become "irrelevant" if they are not used by any current request and fall below some threshold in terms of their "desirability" according to some known eviction policy such as "least recently used" or "greedy-dual".

Step-2, Admission of requests: After the removal step, we load files into the available space in the cache such that the number of requests in the admission queue that can be serviced is maximized. In the weighted version of the problem, each request may have some value reflecting its popularity or priority and the goal in that case is to maximize the total value of serviced requests.

From these two steps, the more interesting for us is Step-2 since Step-1 can be performed based on any known efficient algorithm for file replacement in the cache. The problem presented by Step-2 is called the *Multi-File Caching (MFC)* problem and is described more precisely in Section 2. We will next illustrate the problem with a small example.

1.3 Example

As a small example of the non-weighted version of this problem, consider the bipartite graph shown in Fig. 1. The top set of nodes represents requests and the bottom set of nodes represents files. Each request is connected by an edge to each of the files it needs. Assuming all files are of the same size, each request has value 1, and the cache has available space to hold at most 3 files, it can be shown that the optimal solution of value 3 is achieved by loading the files a, c and e into the cache and servicing the three requests 1,3, and 5. Loading a, b and c has a value of 1 as only request 2 can be served whereas loading b, c and e has a value of 2 as it allows us to serve requests 2 and 5.

1.4 Main Results

The main results of this paper are:

1. Identification of a new caching problem that arises frequently in applications that deal with vertically partitioned files.
2. Derivation of heuristic algorithms that are simple to implement and take into account the dependencies among the files.

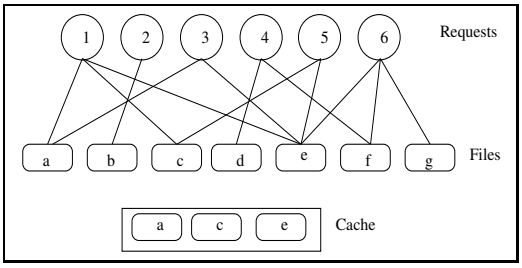

Fig. 1. A bipartite graph depiction of a set of jobs and their file requests

3. Analysis of the heuristics and derivation of tight bounds from the optimal solution
4. Extensive simulation results comparing the new algorithm with the traditional first come first serve.

The rest of the paper is organized as follows. In Section 2 we formally describe the MFC problem and discuss its complexity. In Section 3 a heuristic greedy algorithm, called *Greedy Request Value (GRV)* is proposed and its bounds from the optimal solution are shown using Linear Programming (or LP) relaxation. In Section 4, we present a simulation framework for evaluating the performance of the proposed *GRV* algorithm. Results of the simulation studies, i.e., workload characterization and measurements of performance metrics are presented in Section 5. Conclusions and future work are presented in Section 6.

2 Related Problems and Approximation Complexity

The Multi-File Caching (MFC) problem is defined as follows: Given a collection of requests $R = \{r_1, r_2, \ldots, r_n\}$, each with associated value $v(r_i)$, defined over a set of files $F = \{F_1, F_2, \ldots, F_m\}$, each with size $s(F_i)$ and a constant M, find a subset R' of the requests, $R' \subseteq R$, of maximum total value such that the total size of the files needed by R' is at most M. It is easy to show that in the special case that each file is needed by exactly one request the MFC problem is equivalent to the knapsack problem. The MFC problem is NP-hard even if each request has exactly 2 files. This is done by reduction from the Dense $k-$subgraph (DKS) problem [9]. An instance of the DKS problem is defined as follows: Given a graph $G = (V, E)$ and a positive integer k, find a subset $V' \subseteq V$ with $|V'| = k$ that maximizes the total number of edges in the subgraph induced by V'. Given an instance of a DKS problem, the reduction to an instance of MFC is done by making each vertex $v \in V$ correspond to a file $f(v)$ of size 1. Each edge (x, y) in E corresponds to a request for two files $f(x)$ and $f(y)$. A solution to the MFC instance with a cache of size k corresponds to a solution to the instance of the DKS where the k files loaded into the cache correspond to vertices of the subgraph V' in the solution of the DKS instance. We also note that any approximation algorithm for the MFC problem can be used to approximate a DKS problem with the same bound from optimality. Currently the best-known

approximation for the DKS problem [9] is within a factor of $O(|V|^{1/3-\epsilon})$ from optimum for some $\epsilon > 0$. It is also conjectured in [9] that an approximation to DKS with a factor of $(1+\epsilon)$ is NP-hard. It is also interesting to note that in case each request can start its service when at least one of its files is in the cache (but not necessarily all), the problem becomes equivalent to the Budgeted Maximum Coverage Problem (BMC) [10,11]. Using the above terminology, in the BMC problem we are given a budget L (cache size) and the goal is to select a subset of the files $F' \subseteq F$ whose total size is at most L such that the total value of the requests using files from F' is maximized. It turns out that BMC is easier to approximate. In [10], an approximation algorithm is given for the BMC with a factor of $(1 - 1/e)$ from optimal.

3 A Greedy Algorithm and Bounds from Optimality

Next, we will describe a simple greedy algorithm called Algorithm GRV (Greedy Request Value) and later prove the relationship between the request value produced by this algorithm and the optimal one. First we need some definitions. For a file f_i, let $s(f_i)$ denote its size and let $d(f_i)$ represent the number of requests served by it. The adjusted size of a file f_i, denoted by $s'(f_i)$, is defined as its size divided by the number of requests it serves, i.e., $s'(f_i) = s(f_i)/d(f_i)$. For a request r_i, let $v(r_i)$ denote its value and $F(r_i)$ represent the set of files requested by it. The adjusted relative value of a request, or simply its relative value, $v'(r_j)$, is its value divided by the sum of adjusted sizes of the files it requests, i.e.

$$v'(r_j) = \frac{v(r_j)}{\sum_{f_i \in F(r_j)} s'(f_i)}$$

Algorithm GRV below attempts to service requests in decreasing order of their adjusted relative values. It skips requests that cannot be serviced due to insufficient space in the cache for their associated files. The final solution is the maximum between the value of requests loaded and the maximum value of any single request.

3.1 Linear Programming Relaxation

We now proceed to analyze the quality of the solution produced by this algorithm. The MFC problem can be modeled as a mixed-integer program as follows. Let

$$z_i = \begin{cases} 1 \text{ if the file} f_i \text{ is in cache} \\ 0 \text{ otherwise} \end{cases}$$

and let

$$y_j = \begin{cases} 1 \text{ if all files used by } r_j \text{ are in cache} \\ 0 \text{ otherwise} \end{cases}$$

Then the mixed integer formulation,$\mathcal{P}$, of MFC can be stated as:

$$\mathcal{P}: \quad \max \sum_{j=1}^{n} v(r_j) y_j$$

input : A set of n requests $R = \{r_1, \ldots r_n\}$, their values $v(r_j$, a set of n files F, the sets $F(r_i)$, a cache C of size $s(C)$ and the sizes $s(f_i)$ of all files in F.
output: The *solution* - a subset of the requests in R whose files must be loaded into the cache.

Step 0: */* Initialize */*
Solution $\leftarrow \phi$; //set of requests selected
$s(C') \leftarrow \phi$; // $s(C')$ keeps track of unused cache size
Step 1: Sort the requests in R in decreasing order of their relative values and renumber from $r_1, \ldots, r_n$ based on the this order
Step 2:
for $i \leftarrow 1$ ***to*** n **do**
  if $s(C') \geq s(F(r_i))$ **then**
    Load the files in $F(r_i)$ to the cache
    $s(C') \leftarrow s(C') - s(F(r_i))$; // update unused cache size
    Solution $\leftarrow$ *Solution* $\cup\, r_i$; // add request r_i to the solution
  end
end
Step 2: Compare the total value of requests in *Solution* and the highest value of any single request and choose the maximum

Algorithm 1: GRV

subject to

$$y_j - z_i \leq 0, \forall i \in F(r_j), \text{ and } \forall j$$

$$\sum_{i=1}^{m} s(f_i) z_i \leq s(C), \quad z_i \in \{0, 1\}$$

The linear relaxation of this problem,$\mathcal{P}_\infty$, and its associated dual problem, $\mathcal{D}$, are not only easier to analyze but also provide a useful bound for a heuristic solution procedure.

$$\mathcal{P}_\infty : \quad \max \sum_{j=1}^{n} v(r_j) y_j$$

subject to

$$y_j - z_i \leq 0, \forall i \in F(r_j), \text{ and } \forall j$$

$$\sum_{i=1}^{m} s(f_i) z_i \leq s(C), \quad 0 \leq z_i \leq 1.$$

$$\mathcal{D} : \quad \min s(C)\lambda + \sum_{i=1}^{m} \lambda_i$$

subject to

$$\sum_{i \in F(r_j)} \lambda_{ji} = v(r_j) \text{ for } j = 1, 2, \ldots, n \tag{1}$$

$$\lambda s(f_i) + \lambda_i - \sum_{j:f_i \in F(r_j)} \lambda_{ji} \geq 0, \text{ for } i = 1, 2, \ldots, m, \quad \lambda, \lambda_i, \lambda_{ji} \geq 0, \qquad (2)$$

where λ_{ji} are the dual variables corresponding to the first set of primal constraints, λ is the dual variable corresponding to the cache size constraint, and the λ_i's correspond to the last set of constraints bounding the z's to be less than one.

To avoid trivialities, we assume that for each request j, $\sum_{i \in F(r_j)} s(f_i) \leq s(C)$. That is, each request can be addressed from the cache, otherwise we can eliminate such requests in the problem formulation.

Theorem 1. *Let V_{GRV} represent the value produced by Algorithm GRV and let V_{OPT} be the optimal value. Let d^* denote the maximum degree of a file, i.e., $d^* = \max_i d(f_i)$ then*

$$\frac{V_{OPT}}{V_{GRV}} \leq 2d^*$$

Proof Outline: For lack of space, we provide only an outline of the proof. The full version is in [12]. Consider the algorithm $GRV(LP)$ such that it stops with the last request that can only be accommodated partially (or not at all). It then compares the solution produced to the value of the last request that could not be accommodated and outputs the larger of the two solutions. We can also show, from a detailed analysis, that the integral solution produced by the modified $GRV(LP)$ is at least $1/2d^*$ times the optimal solution. Algorithm GRV can be adapted to produce equivalent or a better solution then $GRV(LP)$□

4 The Simulation Framework for Multi-file Caching

To evaluate various alternative algorithms for scheduling jobs in a Multi-File caching environment, we developed an appropriate machinery for file caching and cache replacement policies that compares GRV and FCFS job admissions each in combination with the least recently used (LRU) replacement policy. Although cache replacement policies have been studied extensively in the literature, these only address transfers between computing system's memory hierarchy, database buffer management and in web-caching with no delays. The model for cache replacement assumes instantaneous replacements. That is that the request to cache an object is always serviced immediately and once the object is cached, the service on the object is carried out instantaneous. As a result the literature gives us very simplistic simulation models for the comparative studies of cache replacement policies. Such models are inappropriate in the practical scenarios for MFC. For instance, once a job is selected for service, all its files must be read into the cache and this involves very long delays.

We present an appropriate simulation model that takes into account the inherent delays in locating a file, transferring the file into the cache and holding the file in the cache while it is processed. The sizes of the files we deal with impose

these long delays. We capture these in the general setup of our simulation machinery. The machinery considers the files to exist in various states and undergo state transitions conditionally from state to state, when they are subjected to certain events.

4.1 The States of a File in a Disk Cache

Each file object associated with the cache is assumed to be in some state. If the file has not been referenced at all it is assumed to be in state S_0. When a reference is made to the file (an event which we characterize subsequently), the file makes a transition to state S_1. A file in state S_1 implies that it has been referenced with at least one pending task waiting to use it but has neither been cached nor in the process of being cached. A file is in state S_2 if it has been previously referenced but not cached and there are no pending tasks for the file. A file is in state S_3 if it has been cached but not pinned and it is in state S_4 if space reservation has been made for the file, but it is still in the process of being cached. A file is in state S_5 if it is cached and pinned. Each of these states is characterized by a set of conditions given by the file status, number of waiting tasks, the last time the file was cached, the job identifier that initiated the caching of the file, and the setting of a cache index.

At some intermittent stages, all files in state S_2 that have not be used for a specified time period are flushed from memory. At this stage all accumulated information, such as the number of reference counts accumulated since its first reference, is lost. The file is said to be set back into state S_0. For our simulation runs all files in state S_2 that have not been referenced in the last five days are cleared. Any subsequent reference to the file would initiate a new accumulation of historical information on the references made to the file. The various states of a file is summarized as follows:

S_0: Not in memory and not-referenced.
S_1: Referenced, not cached but has pending tasks.
S_2: Referenced, not cached and has no pending tasks.
S_3: Cached but not pinned.
S_4: Space reserved but not Cached. Caching in progress.
S_5: Cached and pinned.

4.2 The Event Activities of the Disk Cache

A file that is referenced, cached, processed and eventually evicted from the cache is considered to undergo some state changes. The events affecting the state changes of the files are caused by the actions of the tasks that are invoked by the jobs and other related system actions. Jobs that arrive at a host are maintained in either a simple queue or a balanced search tree depending on the algorithm for processing the jobs.

A job scheduling policy is used to select the job whose files are to be cached next. For FCFS, the job in front of the queue is selected next. In the GRV algorithm, we evaluate the selection criterion for all waiting jobs and select the

recommended one based on the potential available cache space. When a job is selected, all its files are then scheduled to be brought into the disk cache. For each such file of a job a task event is initiated by creating a task token that is inserted into an event queue. Each task token is uniquely identified by the pair of values of the job and file identifiers. A task token is subjected to five distinct events at different times. These events are: *Admit-File* (E_0) *Start-Caching* (E_1), *End-Caching* (E_2), *Start-Processing* (E_3) and *End-Processing* (E_4). Two other events are the *Cache-Eviction* (E_5) and the *Clear-Aged-file* (E_6). The special event, *Clear-Aged-file* (E_5), when it occurs, causes the all the information (e.g., history of references to a file) for files that have been dormant for a stipulated period to be deleted. The entire activities within this framework are executed as a discrete event simulation. The activities of the simulation may be summarized by the finite state machine, with conditional transitions. This is depicted as a state transition diagram of Figure 2. The simulation is event driven and the

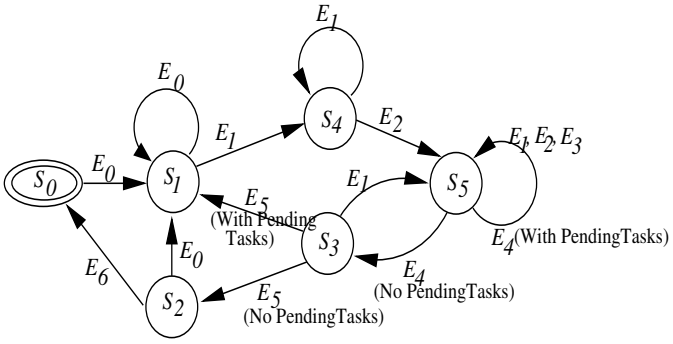

Fig. 2. A Finite State Machine Diagram with Conditional Transitions

three different file processing orders are modeled accordingly with the order of insertions and deletions of files in the data structure T_4.

4.3 File Processing in MFC with Delays

The event selected next is an admission if the arrival time of the next job is earlier than the time of the earliest event. If a job arrival is earliest, it is inserted into the admission structure. On the other hand if the top event of the event queue is earlier, it is removed and processed accordingly. The processing of the event may reinsert an event token into the event queue unless the event is the completion of a task. Each time a job completes, we determine the potential available free cache space and schedule the next job to be processed. In GRV, the potential available cache space is used in the algorithm and not the actual free space. The *potential available free cache space* is the sum of unoccupied space and total size of all unpinned files.

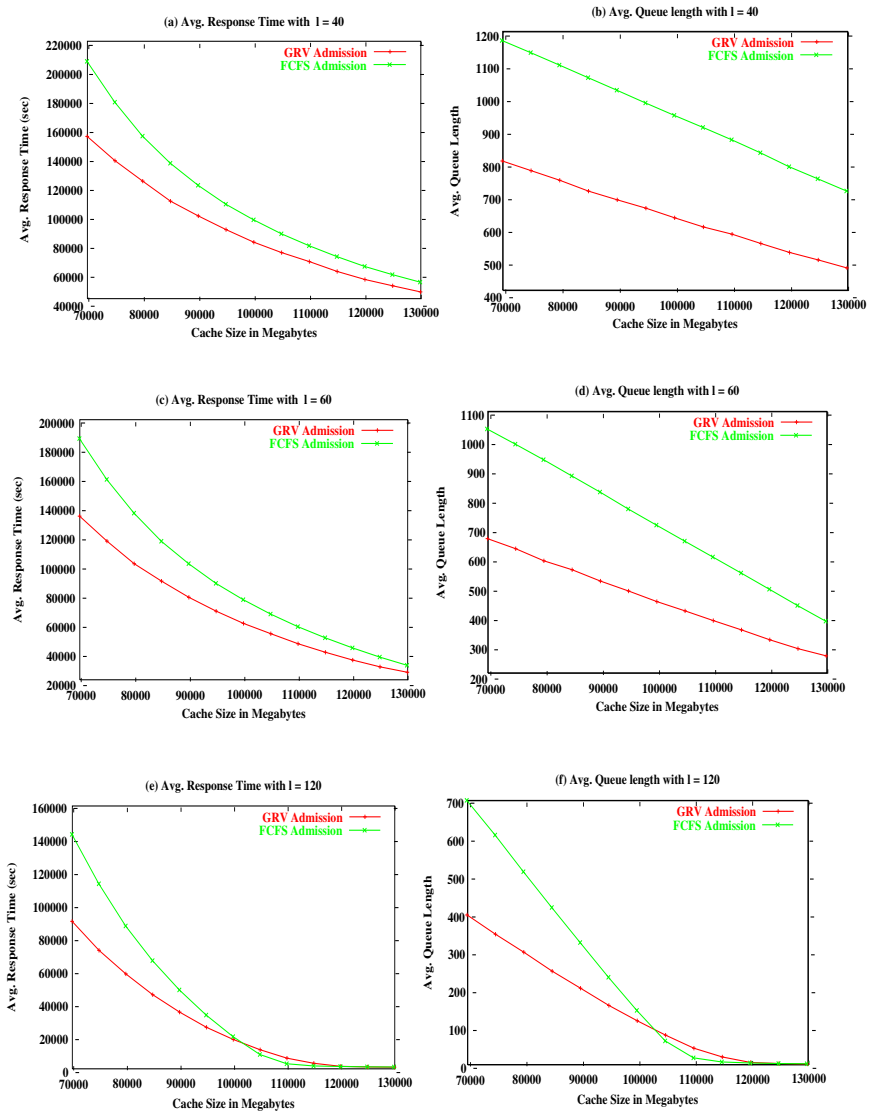

Fig. 3. Graphs of GRV vs FCFS in Multi-file Caching for Average Response Times and Average Queue Lengths

We evaluate the jobs admission policy and the cache replacement policy separately but not independently. In evaluating GRV, a file in the cache is assigned a zero file size and in evaluating a replacement policy, all files that appear in the batch of files to be cached are first pinned by restricting them from being candidates for eviction.

5 Experimental Setup

We conducted experiments using the simulation framework described above to compare the GRV algorithm with a naive FCFS job scheduling of the Multi-file Caching problem when the cache replacement algorithm is LRU. Two performance metrics were used: the average response time of a job and the average queue length jobs with workloads of varying jobs arrival rates. Our implementation is a straight forward translation of the *Finite State Machine (FSM)*, with conditional transitions, to a C++ code. When all the files of a job are cached the tasks associated with the jobs, process the files at a steady rate of 10 MBytes per second. This implies that the processing time of a job is the time to process the file with the largest size.

5.1 Workload Characteristics and Simulation Runs

We subjected the simulation model to workloads in our experiments where the job inter-arrival times are exponentially distributed with mean inter-arrival times of 40, 60 and 120 seconds. Each job makes a request for n files where n is a uniform number between 1 and 25. The file sizes are also uniformly distributed between 500KB and 4GB.

The simulation runs were carried out on a Redhat Linux machine with 512 MBytes of memory. We evaluated the performance metrics of the average response time per job and the average queue length when the cache replacement policy is LRU. For each configuration and for each workload, a number of runs were done with cache sizes varying from 70 to 130 Gigabytes. For each run and for each cache size, we applied a variance reduction method by averaging the statistics that we compute independently for 5 segments of the workload.

5.2 Discussion of Results

Figures 3a, 3c and 3e show the graphs of the response times of synthetic workloads with respective mean inter-arrival times of 40, 60 and 120 seconds. These graphs indicate that the GRV clearly gives a better response time than a simply FCFS job scheduling. GRV performs even better for higher arrival rates. However, as the disk cache sizes increase the graphs of the two algorithms converge.

The graphs of the average queue length shown in Figures 3b, 3d and 3f show similar trends as the graphs of the average response times. This was expected since the average queue length is strongly correlated with the response time for a fixed arrival rate. FCFS admission policy cannot be discarded entirely. As Figures 3e and 3f illustrate, for sufficiently low rate of arrivals and significantly large disk cache size, FCFS job scheduling can perform competitively with GRV. Using different cache replacement policies, e.g., greedy dual size, the same relative results are likely to be achieved. This is left for future work.

6 Conclusions and Future Work

We have identified a new type of caching problem that appears whenever dependencies exist among files that must be cached. This problem arises in various scientific and commercial applications that use vertically partitioned attributes in different files. Traditional caching techniques that make caching decisions one file at a time do not apply here since a request can only be processed if all files requested are cached. Since the problem of optimally loading the cache to maximize the value of satisfied requests is NP hard, we settled on approximation algorithms that were shown analytically to produce solutions bounded from the optimal one. The MFC problem is also of theoretical interest in its own right because of its connection to the well known dense k-subgraph and the fact that any approximation to MFC can be used to approximate the latter problem with the same bounds from optimality.

The simulation studies show that the new algorithms outperform ($FCFS$) scheduling of requests with multiple file caching. The system throughput using schedules based on Algorithm GRV is consistently higher than the FCFS schedules and the queue length is correspondingly shorter. Future work to be conducted involves detailed simulations using real workloads derived from file caching activities of data intensive applications. We also intend to pursue these studies for both synthetic and real workloads for multiple file replacements at a time rather than one file at a time.

Acknowledgment

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. Cao, P., Irani, S.: Cost-aware WWW proxy caching algorithms. In: USENIX Symposium on Internet Technologies and Systems. (1997)
2. Hahn, U., Dilling, W., Kaletta, D.: Adaptive replacement algorithm for disk caches in hsm systems. In: 16 Int'l. Symp on Mass Storage Syst., San Diego, California (1999) 128 – 140
3. Otoo, E.J., Rotem, D., Shoshani, A.: Impact of admission and cache replacement policies on response times of jobs on data grids. In: Int'l. Workshop on Challenges of Large Applications in Distrib. Environments, Seatle, Washington, IEEE Computer Society, Los Alamitos, California (2003)
4. Tan, M., Theys, M., Siegel, H., Beck, N., Jurczyk, M.: 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, Orlando, Florida (1998) 115–129
5. Wang, J.: A survey of web caching schemes for the Internet. In: ACM SIGCOMM'99, Cambridge, Massachusetts (1999)

6. Young, N.: On-line file caching. In: SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms). (1998)
7. Shoshani, A.L.B., Nordberg, H., Rotem, D., Sim, A.: Multidimensional indexing and query coordination for tertiary storage management. In: Proc. of SSDBM'99. (1999) 214 – 225
8. Wu, K., Koegler, W.S., Chen, J., Shoshani, A.: Using bitmap index for interactive exploration of large datasets. In: SSDBM'2003, Cambridge, Mass. (2003) 65–74
9. Feige, U., Peleg, D., Kortsarz, G.: The dense k-subgraph problem. Algorithmica **29** (2001) 410–421
10. Khuller, S., Moss, A., Naor, J.S.: The budgeted maximum coverage problem. Information Processing Letters **70** (1999) 39–45
11. Krumke, S.O., Marathe, M.V., Poensgen, D., Ravi, S.S., Wirth, H.C.: Budgeted maximum graph coverage. In: int'l. Workshop on Graph Theoretical Concepts in Comp. Sc., WG 2002, Cesky Krumlov, Czech Republic (2002) 321 – 332
12. Rotem, D., Seshadri, S., Otoo, E.J.: Analysis of multi-file caching problem. Technical report, Lawrence Berkeley National Laboratory (2003)