File Caching in Data Intensive Scientific Applications

Ekow Otoo, Doron Rotem and Alexandru Romosan
Lawrence Berkeley National Laboratory,
University of California,
Berkeley, California 94720

Sridhar Seshadri
Leonard N. Stern School of Business,
New York University, 44 W. 4th St., 7-60,
New York, 10012-1126

Abstract

We present some theoretical and experimental results of an important caching problem which arises frequently in data intensive scientific applications. For such applications, jobs need to process several files simultaneously, i.e., a job can only be serviced if all its needed files are present in the disk cache. The set of files requested by a job is called a file-bundle. This requirement introduces the need for cache replacement algorithms based on file-bundles rather then individual files. We show that traditional caching algorithms such Least Recently Used (LRU) *and* GreedyDual-Size (GDS) *are not optimal in this case since they are not sensitive to file-bundles and may hold in the cache non-relevant combinations of files. We propose and analyze a new cache replacement algorithm specifically adapted to deal with file-bundles. We tested the new algorithm using a disk cache simulation model under a wide range of conditions such as file request distributions, relative cache size, file size distribution, and incoming job queue size. In all these tests, the results show significant improvement over traditional caching algorithms such as GDS.*

1. Introduction

1.1. Overview

Data intensive scientific applications concern application software that have very large data and storage resource requirements. Such applications are becoming increasingly prevalent in domains of scientific and engineering research, examples include long running simulations of time-dependent phenomena that periodically generate snapshots of their state as in Astrophysics and climate modeling, simulation of combustion phenomena, and very large data sets generated from experiments such as BaBar [2] or the Large Hadron Collider (LHC) in the area of high energy particle physics. The large datasets, from simulations and actual experiments, are preprocessed and maintained in units of files on geographically dispersed mass storage systems. Subsequent data analyses and visualization applications retrieve subsets of these files onto high performance computing resources creating large demands for disk and tape storage retrieval and network bandwidth.

Caching has long been recognized as one of the most important techniques to reduce bandwidth consumption [1, 3, 9, 10]. The general use of the term caching implies a specialized buffer storage that is used to speed up access when the data is transferred between different levels of a storage hierarchy with different characteristics: speed of access, size and cost per bit (see Fig. 1).

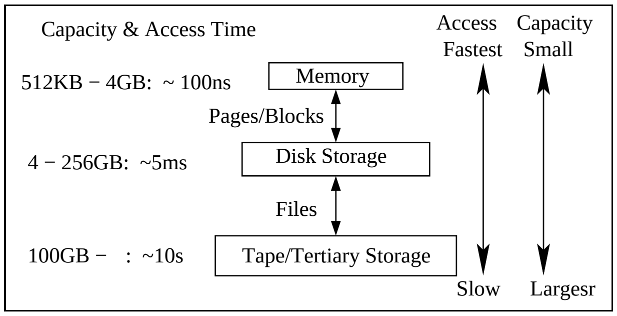


Figure 1. The Different Levels of Caching in Data Intensive Applications

Successful caching relies on two properties of the access patterns of most application to be effective:

temporal locality - if a file is accessed once, it is likely to be accessed again soon

spatial locality - if a file is accessed then files in close proximity (e.g., on the same storage tape) are also likely to be accessed.

The disk cache manager often has no knowledge of the request stream for files in the cache. Consequently, many cache systems are based on recognition of patterns of either recently used or frequently used files and use this to determine which files should be kept in cache and which should be evicted.

1.2. Problem description and Previous Work

Consider a sequence of jobs that make requests for files at a computational resource where each job is comprised of one or more file requests. The requests are serviced in some order: *first come first serve (FCFS), shortest job first (SJF),* etc. A cache C of some fixed size $s(C)$, is available for storing a subset of all the requested files. A job is serviced only if all the files it needs are already in the cache C, otherwise it waits in queue until all its requested files are transferred from a Mass Storage System (located either locally or at a remote site) into C. These data transfers cause time delays for the job execution, as well as consumption of valuable resources such as network and data storage bandwidth. The problem we address is therefore finding an optimal cache replacement policy that maximizes throughput, or alternatively minimizes the volume of data transfers, under a limited cache space.

We refer to the set of files requested by a job as a *file-bundle*. Processing a job requires that all the files in its *file-bundle* be present simultaneously in the cache. For this reason, it is necessary in this environment to make cache loading and replacement decisions based on *file-bundles* rather then a single file at a time as in traditional file caching algorithms. This difference is quantified further in section 3.

The quest for optimal caching strategies has posed some interesting challenges and has culminated in the development of numerous cache replacement policies some of which include: Least Recently Used (LRU), Least Frequently Used (LFU), Greedy Dual Size (GDS) and Minimum Average Cost Per Replacement (MACR). The closest environment, in the use of caching techniques, to the one desirable in scientific data management, is in web-caching where proxy servers and reverse proxy servers are configured essentially as distributed caches. The most widely used caching system in this domain is the Squid Cache System. Other distributed systems that provide caching functionalities are the *dCache* [5], and Storage Resource Managers (SRM) [11].

Although the literature provides a considerable number of papers [4, 7, 8, 12, 14, 16] that describe and analyze caching and replacement policies, the main concern of most of these efforts is the maintenance of a “popular” set of files in the cache in order to maximize “hit” ratios and minimize expected access costs for requested files not found in cache. The problem discussed in this paper is radically different from these earlier works as requests require caching of multiple files simultaneously rather than single files.

We propose here algorithms based on an analysis of the problem that maximizes the throughput of jobs, i.e., number of jobs serviced per unit time, while also minimizing the *byte miss ratio*. The typical performance metrics in cache replacement algorithms are the *hit ratio*, the *miss ratio*, the *byte hit ratio*, and the *byte miss ratio*. A good cache replacement policy maximizes the *hit ratio* (or minimizes the *miss ratio*) or alternatively maximizes the *byte hit ratio* (or minimizes *byte miss ratio*). Suppose we have a workload of a sequence of N jobs $R = \langle r_1, r_2, \dots r_N \rangle$, where each job $r_i = \{f_i\}$ makes a request for only one file f_i . Let the size of a file f_i by denoted as $s(f_i)$. Of the N requests, let the set of files found in the cache be H where h is its cardinality, i.e., $h = |H|$. The hit ratio ρ_{hit}, is defined as $\rho_{hit} = h/N$. The miss ratio ρ_{miss} is defined as $1 - \rho_{hit} = 1 - h/N$. The byte hit ratio $\rho_{byte-hit}$ is defined as $\rho_{byte-hit} = (\sum_{i \in H} s(f_i))/(\sum_{j \in R} s(f_j))$ and the byte miss ratio $\rho_{byte-miss}$ is defined as $1 - \rho_{byte-hit}$.

We compare our results with some earlier works on caching, using the *byte miss ratio* as our performance metric for most of the experiments. We also show how the results are affected when queues of waiting jobs are taken into consideration. The rational in choosing *byte miss ratio* (or conversely, the byte hit ratio), metric for comparison is that, we wish to minimize the amount of data transfered into and out of the cache. When considering single file caching, there is a strong correlation between the *byte miss ratio* and the *byte hit ratio* but this is not necessarily true, when file-bundles are considered.

1.3. Main Results

The main results of this paper are:

- Identification of a new caching problem, which arises frequently in scientific applications that deal with *file-bundle* caching.
- Derivation of a new cache replacement algorithm *File Bundle Cache (FBC),* that is simple to implement. Unlike existing cache replacement algorithms in the literature, we track the *file-bundles* that were requested in the past to determine what combinations of files should be retained or evicted from the cache. This results in a much lower cache miss-ratio under a wide range of conditions tested.
- Results of extensive simulation runs that compare the *FBC* algorithm with *GreedyDual-Size* [4] cache replacement consistently show that *FBC* gives a much lower average volume of data transfers per request with file requests observing either Uniform or Zipf distributions.
- The heuristic algorithm *OptCacheSelect* used by *FBC* is an approximation algorithm to an interesting combinatorial problem whose exact solution is NP-Hard. For this algorithm, we derive tight bounds from the optimal solution and show that the value of the solution produced is a factor of at most $1/(2d^*)$ of the optimal one where d is the maximum number of requests that use the same file.

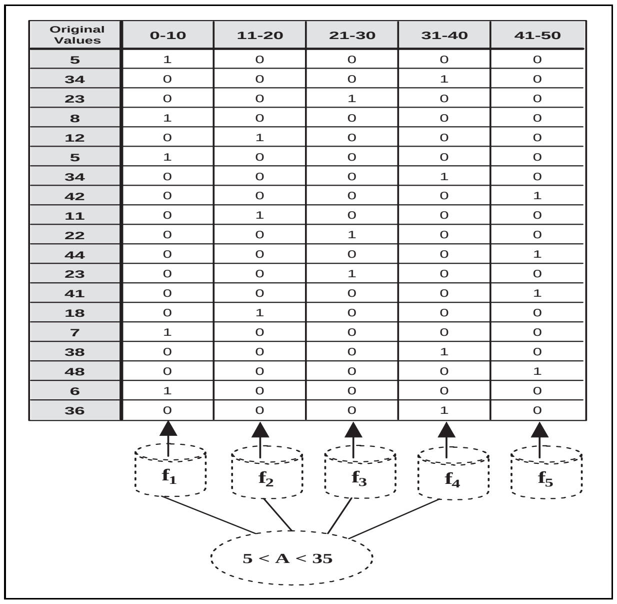
The rest of the paper is organized as follows. In Section 2 we discuss file caching and its significance to data intensive application. In Section 3 we present a heuristic based on a greedy algorithm called *FBC*, and its bound from the optimal solution is derived using LP relaxation in Section 4. Our experimental set for the comparison of our proposed algorithm with GredyDual-Size is introduced in Section 5 and the results are discussed in Section 6. We conclude with Section 7 where we give some directions for future work.

2. File Caching in Scientific Data Management

This work is motivated by file caching problems arising in scientific and other data management applications that involve multi-dimensional data [8, 15]. The main common characteristic of such applications is that they deal with objects that have multiple attributes (10 to 500), and often partition the data such that values for each attribute (or a group of attributes) are stored in a separate file (vertical partitioning). Subsequent analysis and data mining jobs that operate on this data often require that several of these attributes are compared or combined together for further computation. In relational database terminology, this is equivalent to computing a multi-way join.

An example of caching of *file-bundles* comes from the area of bitmap indices for querying high dimensional data [15]. In this case, a collection of N objects (such as physics events) each having multiple attributes, is represented using bitmaps in the following way: the range of values of each attribute is divided into sub-ranges also called bins; a bitmap is constructed for each bin with a ‘0’ or ‘1’ bit indicating whether an attribute value is in the required subrange (see Fig. 2). The bitmaps (each consisting of N bits before compression) are stored in multiple files, one file for each bin of an attribute. Range queries are then answered by performing boolean operations among these files. Again, in this case all files containing bitmaps relevant to the query form a *file-bundle* as they must be read simultaneously to answer the query. As a small example, in Fig. 2 the values of attribute A are partitioned into 5 bins each stored in a separate file. A request that includes a range query $5 \leq A \leq 35$ requires caching of a *file-bundle* consisting of the files f_1, f_2,f_3, and f_4. These files must be cached simultaneously in order to perform an “OR” operation on them.

Other examples of applications that require *file-bundles* are applications that need to compute derived data based on raw data residing in several files. For example applications that analyze physics experimental data coming from detectors, require *file-bundles* con-



Original Values	0-10	11-20	21-30	31-40	41-50
5	1	0	0	0	0
34	0	0	0	1	0
23	0	0	1	0	0
8	1	0	0	0	0
12	0	1	0	0	0
5	1	0	0	0	0
34	0	0	0	1	0
42	0	0	0	0	1
11	0	1	0	0	0
22	0	0	1	0	0
44	0	0	0	0	1
23	0	0	1	0	0
41	0	0	0	0	1
18	0	1	0	0	0
7	1	0	0	0	0
38	0	0	0	1	0
48	0	0	0	0	1
6	1	0	0	0	0
36	0	0	0	1	0

Figure 2. Illustration of file-bundle consisting of bitmap files

sisting of files with measurement data (energy level, momentum etc.) together with other files containing instrument calibration data for proper interpretation of the measurements.

3. Algorithms and Bounds from Optimality

3.1. File Bundle Caching Algorithm

The main idea behind our caching strategy is to load the cache with a set of files that correspond to popular *file-bundles*, thus maximizing the probability that an arriving request can find all the files it needs in the cache. We illustrate the difference between this strategy and caching policies based on single file popularity with a small example shown in Fig. 3: for a given cache state and a request r, we say that the cache supports r or, alternatively, that r is a request-hit if the *file-bundle* needed by r is found in the cache.

3.2. Example

Let us assume that we have six possible requests $r_1, r_2, ..., r_6$ each associated with a *file-bundle* drawn from $F = f_1, f_2, ...f_7$ as shown in Table 1 and in Fig. 3 by the lines connecting requests to their associated files. Further, let us assume that all files are of the same size, the cache can hold only three files, and all six requests are equally likely, *i.e.* with a probabilty of $\frac{1}{6}$ that any request is the next one to arrive. Each row in Table 2 shows the probablity of the event that a file is requested by a random request. Note that the sum of probabilities is more than 1 as the events are not mutually exclusive. We note that the most popular file is f_5 as 4 requests out of the six possible requests need it. This is followed by files f_6 and f_7 each needed by 3 of the requests. Each row in Table 3 shows request-hit probabilities, *i.e.,* the probablity that a random request will find its *file-bundle* in the cache under some cache content. Only 5 cases of cache content out of the 35 cases (possible ways of choosing 3 files from 7) are shown. We note that keeping the 3 most popular files (row 1 of the table) does not lead to the largest request-hit probability. The best request-hit probability is represented in the cache of Fig. 3 and by the second row of the table with a request-hit probability of $\frac{1}{2}$ as keeping files f_1, f_3, f_5 in the cache results in a request-hit for 3 out of the six possible requests.

Request	File-Bundle
r_1	f_1,f_3,f_5
r_2	f_2,f_6,f_7
r_3	f_1,f_5
r_4	f_4,f_6,f_7
r_5	f_3,f_5
r_6	f_5,f_6,f_7

Table 1. Requests and their file-bundles

File	No of Requests	File request probability
f_1	2	1/3
f_2	1	1/6
f_3	2	1/3
f_4	1	1/3
f_5	4	2/3
f_6	3	1/2
f_7	3	1/2

Table 2. File request probabilities

The previous example illustrates the need for caching strategies that take into account request-hits rather than simple file-hit based algorithms. We also note that a simplistic approach that loads or evicts *file-bundles* from the cache associated with requests based solely on their popularity (LFU- Least Fre-

Cache Contents	Requests Supported	Request-hit probability
f_5,f_6,f_7	r_6	1/6
f_1,f_3,f_5	r_1,r_3,r_5	1/2
f_1,f_5,f_6	r_3	1/6
f_3,f_5,f_6	r_5	1/6
f_1,f_2,f_3	-	0

Table 3. Request-hit probabilities

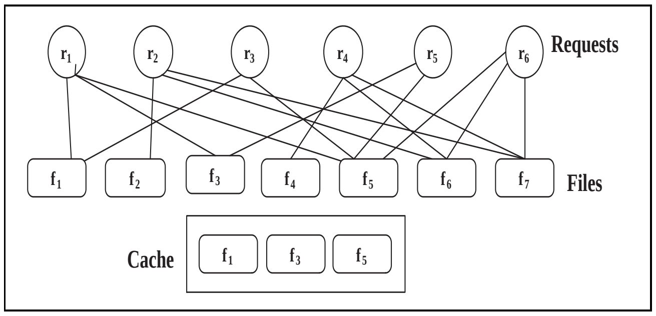


Figure 3. Example of file selection

quently Used based algorithms) does not work, as file-sharing between *file-bundles* must also be taken into account. For example, let us consider a small subset of the requests as shown in Fig. 4. The relative popularity of each of the 3 requests is also given. Eviction of the *file-bundle* associated with the relatively unpopular request r_2 (popularity of .2) will cause a cache miss for the highly popular requests r_1 and r_3 (each with popularity of .4) whose *file-bundles* overlap with it. Similar example can be constructed for *file-bundle* LRU based algorithms as well. For that reason, the degree of file-sharing must also be taken into account by an effective cache replacement algorithm.

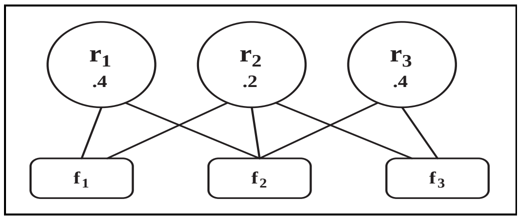


Figure 4. Example file-bundle overlaps

We now proceed to describe our caching algorithm, *FBC*, which loads and evicts files from the cache in response to new requests. At the heart of our caching strategy is an algorithm called *OptCacheSelect*, called by *FBC*, to determine which files must be loaded and/or replaced.

We will first describe *OptCacheSelect* and then show how it is incorporated into the main algorithm *FBC*. It takes into account file sizes and request frequency counts as well as degree of file-sharing. It will be described in more detail below. The result produced by *OptCacheSelect* is a new set of files loaded into the cache that attempts to maximize request-hit probability. The algorithm is a greedy heuristic that attempts to achieve a good approximation to an NP-hard problem that is a generalization of the Knapsack problem. Some theoretical results about the complexity of the problem and analysis of the effectiveness of the approximation are given in Section 4

The *OptCacheSelect* algorithm gets as its input a data structure $L(R)$ containing full information about a collection of historical requests R. The data structure $L(R)$ is initially empty and gets updated with each request processed. For lack of space we will not present here the exact implementation of $L(R)$, which is basically a hash-table with pointers to other structures, but rather describe its contents. For each request $r_i \in R$ that was served by the system we store in $L(R)$ the following information:

- An associated value $v(r_i)$. In our current implementation $v(r_i)$ is simply a counter incremented by 1 each time this request appeared so far, but it can also reflect request priority or some other measure of importance. In Section 6 we show how this function can be used to enhance "fair" scheduling of the requests.
- the set $F(r_i)$ of files requested by r_i and the size of each such file.

We need the following additional definitions in the description of the algorithm. We denote the size of a cache C by $s(C)$. 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., $s'(f_i) = s(f_i)/d(f_i)$.

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.*

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

The algorithm *OptCacheSelect($L(R)$,$S(C)$)* attempts to select an optimal set of files that fits in the cache in order to serve a subset of R with the highest total value. It does so by servicing requests in decreasing order of their adjusted relative values skipping 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. The justification for the comparison performed in this latter step is given in Appendix A.

The intuition behind using $v'(r_j)$ as a measure for ranking requests is that $v'(r_j)$ increases with an increase in request popularity and degree of sharing of its files with other requests. On the other hand, it decreases when the amount of cache resources used by $F(r_j)$ grows.

```
input  : A data structure $L(R)$ as described
         above and a cache $C$ of size $s(C)$
output: The solution $G$ - a subset of the
         requests in $R$ whose files must be
         loaded into the cache.

Step 0: /* Initialize */
$G \leftarrow \phi$;    //set of requests selected
$s(C') \leftarrow s(C)$ ;   // $s(C')$ keeps track of
unused cache size
Step 1: Sort the requests in $R$ in decreasing
order of their relative values and renumber
from $r_1, \dots, r_n$ based on this order
Step 2:
for $i \leftarrow 1$ to $n$ do
    if $s(C') \geq s(F(r_i))$ then
        Load the files in $F(r_i)$ into the cache
        $s(C') \leftarrow s(C') - s(F(r_i))$ ;   //
        update unused cache size
        $G \leftarrow$ G $\cup\, r_i$ ;    // add request $r_i$ to
        the solution
    end
end
Step 3: Compare the total value of requests
in $G$ and the highest value of any single
request and choose the maximum.
```

Algorithm 1: Algorithm OptCacheSelect

Note: In practice we can even do better by recomputing $v'(r_j)$ for all requests r_j not selected yet (and resorting) following Step 2. This is done by setting to 0 the size of files in $F(r_j)$ that are already in the cache. The reason for this is that these files will not consume any additional cache resources. This leads to an increase in adjusted value for requests that share files with the previously selected requests.

We are now in a position to describe the main steps of our caching algorithm, *FBC*, as illustrated in Fig. 5. Initially the cache is empty, whenever a new request r_{new} arrives all its missing files (files requested by it but not currently in the cache) are loaded into the cache (Fig. 5a). At some point the cache fills up (Fig. 5b) and a caching replacement decision must be taken when a new request, r_{new}, arrives.

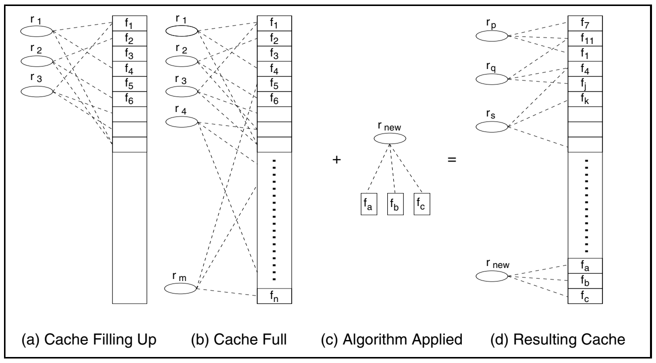


Figure 5. The steps of algorithm FBC

All files requested by r_{new} not currently present in the cache must be loaded into the cache and some other files currently in the cache must be evicted in order to make space for them (Fig. 5c). We reserve sufficient space for the new files requested by r_{new} and then call on algorithm *OptCacheSelect* described above to decide on the optimal files that must be maintained in the remaining part of the cache to maximize request-hit probability (Fig. 5d).

The Algorithm 2 formalizes the steps of the *FBC* algorithm.

4. Complexity Analyses of the Algorithms

4.1. Linear Programming Relaxation

We now proceed to analyze the quality of the solution produced by the OptCacheSelect algorithm which is at the heart of our caching strategy. We refer to the problem faced by this algorithm as the File-Bundle Caching (FBC) problem which is defined as follows: Given a collection of requests $R = \{r_1, r_2, \dots, r_n\}$,

```
input  : A new request $r_{\text{new}}$, a data structure
         $L(R)$ including information about
         requests $R = \{r_1, \ldots r_n\}$, their
         values $v(r_j)$, the sets $F(r_i)$, a cache
         $C$ of size $s(C)$ ,$F(C)$ the set of files
         currently in the cache, and the sizes
         $s(f_i)$ of all files requested by
         members of $R$.
output: The solution $G$ - a set of files that
         must be loaded into $C$

Step 1: Compute $S$, the amount of space
needed by files in $F(r_{new})$ that are not
currently in the cache $C$
Step 2: Call OptCacheSelect(L(R),s(C)-S)
and store its solution in $F(Opt)$
Step 3: Load into the cache $C$ the files in
$F(Opt) \backslash F(C)$
Step 4: Update the data structure $L(R)$ with
all relevant information about $r_{\text{new}}$
```
Algorithm 2: Algorithm OptFileBundle

and a constant M. Each request in R is associated with a value $v(r_i)$ and a subset of files (*file-bundle*) drawn from a set of files $F = \{F_1, F_2, \ldots, F_m\}$ where each file in F has a size $s(F_i)$. 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 *file-bundles* have no overlaps, i.e., each file is needed by exactly one request the FBC problem is equivalent to the knapsack problem. The FBC problem is NP-hard even if *file-bundles* contain exactly 2 files. This is done by reduction from the Dense $k-$subgraph (DKS) problem [6]. 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 FBC 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 a *file-bundle* consisting of the two files $f(x)$ and $f(y)$. A solution to the FBC 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 FBC problem can be used to approximate a DKS problem with the same bound from optimality. Currently the best-known approximation for the DKS problem [6] is within a factor of $O(|V|^{1/3-\epsilon})$ from optimum for some $\epsilon > 0$. It is also conjectured in [6] that an approximation to DKS with a factor of $(1+\epsilon)$ is NP-hard.

In view of the complexity of approximating FBC problem, we now derive a bound using LP relaxation techniques on the ratio between the value of an optimal solution to the FBC problem and the one produced by algorithm OptCacheSelect. The best bound we are able to derive is $2d^*$ where d^* represents the maximum number of requests sharing a single file. This is summarized in the following theorem

Theorem 4.1. *Let $V_{OptCachSelect}$ represent the value produced by Algorithm OptCachSelect 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*

$$\frac{V_{OPt}}{V_{GR}} \leq 2d^*$$

The problem formulation of the LP, its dual and the details of the proofs are presented in Appendix A.

5. Simulation Framework

We designed a simulation model to explore how the *FBC* algorithm compares with the *GreedyDual-Size* algorithm [4, 16]. For that purpose, we implemented a modified version of *GreedyDual-Size* where each request is for a set of files rather than a single file as in the original implementation. The implementation is described below in Algorithm 3. Each cached file f is associated with a value $H(f)$ defined by the cost $c(f)$ of reading the file into cache, and the size $s(f)$ of the file. Whenever space is required, the file selected for eviction is that with the minimum $H(f)$ value. By maintaining cached files as *min-heap* data structure based on the $H(f)$ values, the selection of candidates for eviction is done in logarithmic time. The modification to the *GreedyDual-Size* to *GreedyDual-MF* is conducted by repeated eviction of the file q which has the minimum H value until all the files in the new request can be accommodated in the cache.

```
Input: A request stream $R = r_1, r_2, ....N$, a
       cache $C$ of size $s(C)$, $F(C)$, the set
       of files currently in cache. Note: a
       request is for a set of files
       $r_i = \{f_0, f_1, \ldots, f_{r_i}\}$
Result: Loading of the cache that satisfies
        GreedyDual-Size Algorithm for each
        request $r_i$.
Initialize $L \leftarrow 0$ ;
for $i \leftarrow 1$ to $N$ do
    Get next request
    Set $req \leftarrow r_i$ ;
    foreach $f \in req$ do
        if f is already in cache C then
            $H(f) \leftarrow L + c(f)/s(f)$ ;
        else
            while there is not enough room in
            $C$ to hold $f$ do
                Set $L \leftarrow min_{\substack{q \in C \\ q \notin r_i}} H(q)$ ;
                Evict q such that $H(q) = L$ ;
                Load f into the cache $C$ ;
                Set $H(f) \leftarrow L + c(f)/s(f)$ ;
            end
        end
    end
end
```

Algorithm 3: Algorithm GDS-MF

5.1. Workload Characterization

We constructed a simulated workload consisting of a given set of jobs, with each job requesting a random number of files from a pool of available files. The parameters chosen for our simulated workload are as close as possible to observed real experiments that log single file requests at a time. The size of each file was generated randomly between a minimum size of 10 MB and a maximum size of 300 MB. The set of files requested by each job was chosen uniformly from the list of available files such that the total size of the files requested by any given job was smaller than the available cache size. The cache sized varied between 5 GB and 100 GB. Each simulation run consists of generating a large number of jobs ($\approx$ 20000) and selecting a number of them (typically 10000) using either a uniform or Zipf distribution in order to study the effects of the various parameters.

5.2. Simulation Environment

The simulation program, *cacheSim*, was written in *C++* with extensive use of STL. Using a cluster of three 1.6 GHz dual Opterons with 2GB of RAM each, we ran a large number of experiments to study the behaviour of the proposed algorithms for different combinations of parameters. These experiments consumed over 1000 hours of CPU time. The main performance metric used is the *byte miss ratio* and this was observed primarily for two different workload distributions, and varying cache sizes.

There are several parameters of interest that affect the result of the simulation:

Popularity Distribution. The popularity distribution of requests for typical workloads is very hard to characterize as it varies widely from setup to setup and even from day to day. As such we are looking at the effects of the two extremes: a purely random distribution, and a Zipf one.

Cache Size. Given requests of a known average size, varying the cache size determines the number of requests that can fit in cache at any given time. The more requests already in cache, the more likely that files requested by an incoming job are already present in cache.

Incoming Queue Length. Instead of processing a job as soon as it is submitted, one can also consider aggregating the jobs in an incoming queue of a given length, and only submitting the best job once the queue is full.

Queueing Fairness Requests with a consistently low value stay queued for a large number of iterations. To improve the fairness of the queueing algorithm, we increase the value of a request in queue by a function of the number of iterations the request has been waiting.

6. Discussion of Results

In this section we describe some representative results of our simulation runs where we studied the effects of various caching parameters on the performance of our algorithm. The first set of experiments performed involved job popularity distributions. A uniform popularity distribution means that every request from the pool of available requests is equally likely to be requested, whereas Zipf's distribution assigns a probability of selection proportional to $\frac{1}{i}$ to the i^{th} most popular request.

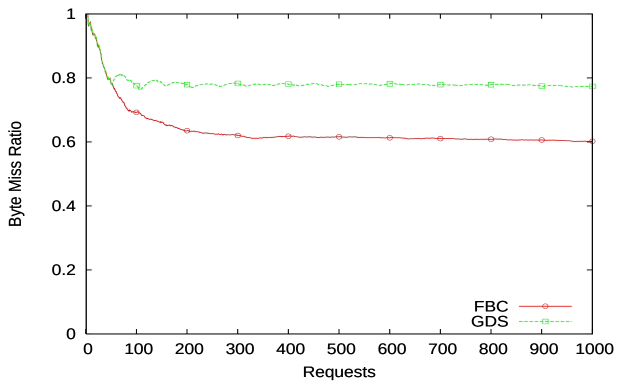
Figures 6(a) and 6(b) compare the *byte miss ratio* for uniform and Zipf request distributions. The cache replacement strategy based on the *FBC* algorithm is superior to the one based on the *GDS* algorithm in the sense that the byte miss ratio is lower. The improved performance of the *FBC* algorithm is attributed to keeping requests coherent in the cache, as well as to keeping track of the popularity of each request. The latter is evident when we consider a Zipf distribution where a small set of requests occurs with a high frequency. The *FBC* algorithm increases the value of each request by keeping track of its popularity thus increasing the likelihood of the request staying in the cache.

The overall effect of varying the cache size on the byte miss ratio is shown in Fig. 6, where the *Relative Cache Size* is defined to be the ratio of the total size of the files requested to the cache size. As the cache is able to serve more requests the amount of data moving into the cache for each request decreases. Three factors contribute to this effect: 1) the number of files common to all the requests already cached increases, thus minimizing the amount of new data that needs to be brought into the cache, 2) the likelihood of often seen requests to be already in the cache increases, and 3) the efficiency of the *FBC* algorithm improves with the number of requests considered when making a decision as to what requests to keep in cache. The first factor dominates the improved byte miss ratio for the uniform distribution shown in Fig. 7(a) while the second one dominates for the Zipf distribution in Fig. 7(b), hence the dramatic improvement in performance with increasing the cache size.

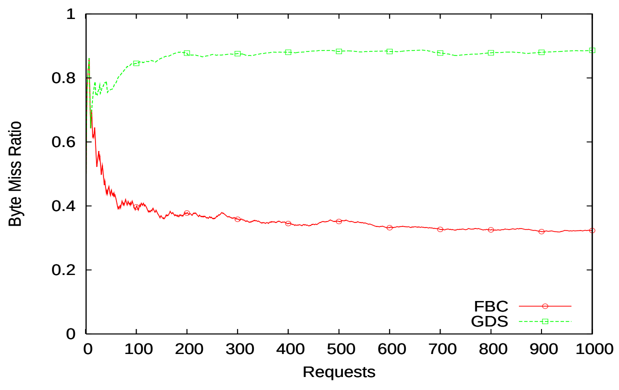
Another set of experiments performed involved aggregating the jobs in a processing queue of varying length instead of processing them in FIFO order. Once the queue is full, we run the *FBC* algorithm on the queued requests, and then process the request with the highest value. Fig. 8(a) and Fig. 8(b) show the effects of varying the processing queue length for a random and Zipf incoming request distribution, respectively. As we increase the queue length the byte miss ratio improves. This is due in part to the increased efficiency of the algorithm with the number of requests considered, in effect translating to an expansion of the cache size.

A negative consequence of queuing requests is that requests with a consistently low value may experience starvation, i.e., stay queued for a large number of iterations. Figures 9(a) and 9(b) show the queueing effects for uniform and Zipf distributions, respectively, for a queue length of 10: while the majority of requests are scheduled right away, there exists a tail of requests which linger in the queue for a large number of iterations. The effect is twofold: first, the queue is in effect shorter thus decreasing the efficiency of the algorithm; second, the execution of such requests is delayed unreasonably which might not be acceptable in a real-life environment. To mitigate the situation we artificially increase the value of a request in queue by an "anti-starvation" function that takes into account the number of iterations *i* the request has been waiting in the queue before being scheduled for execution. Three different functions were considered: increase the request value by 1) i (*1-weight*), 2) i^2 (*2-weight*), and 3) 2^i (*e-weight*). Since the value of a request depends on its popularity, the effect of increasing a request value by *1-weight* or *2-weight* on the wait time is less pronounced for a Zipf distribution than for a uniform one since the natural increase in value with popularity of the most often requested requests is of about the same order of magnitude. Only when we artificially increase the value of a queued request by *e-weight* for a Zipf distribution we begin to see it dominate the natural increase with popularity.

The effect on the byte miss ratio of our various "anti-starvation" functions is shown in fig. 10. Surprisingly, we see that the effect of such functions leads to an improvement in the byte miss ratio, in addition to the expected improvement in the fairness of the system by scheduling all requests in a reasonable number of iterations. Only when we increase the value of a queued request by *e-weight* we observe a degradation in the byte miss ratio. We interpret such an increase to having the effect of delaying the scheduling of all the requests by the queue length and in effect reducing the effective queue length to a value of 1. Further studies need to be performed to understand the tradeoffs presented by different types "anti-starvation" functions.

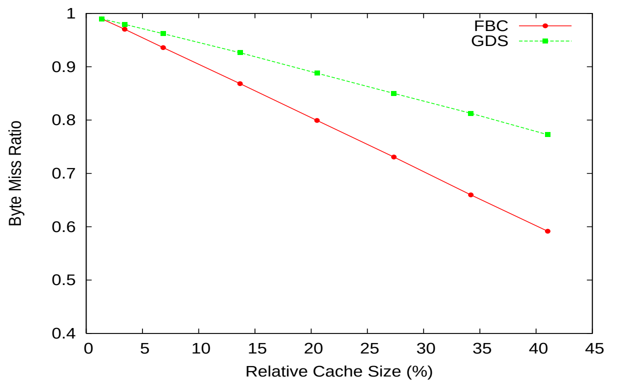


(a) Uniform Distribution.

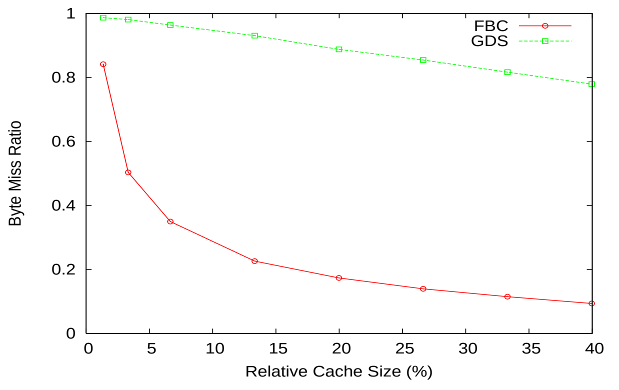


(b) Zipf Distribution.

Figure 6. Byte Miss-Ratio vs. Cache Size

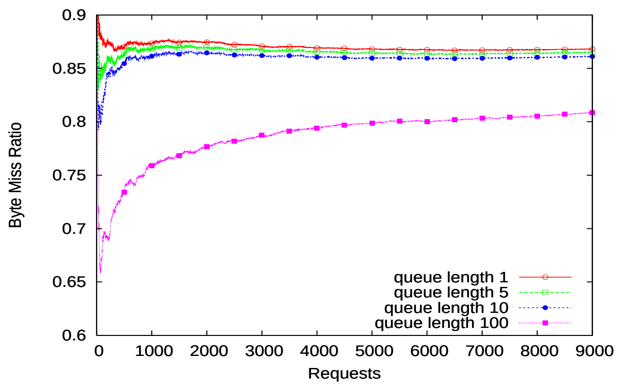


(a) Uniform Distribution.

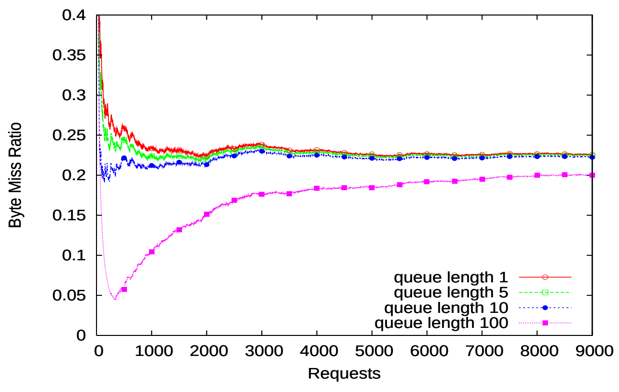


(b) Zipf Distribution.

Figure 7. Effect of Varying Cache Size

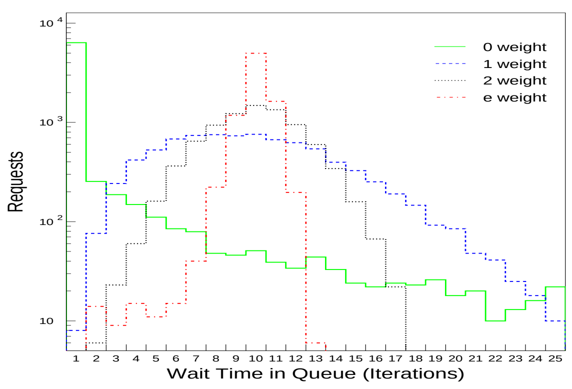


(a) Uniform Distribution.

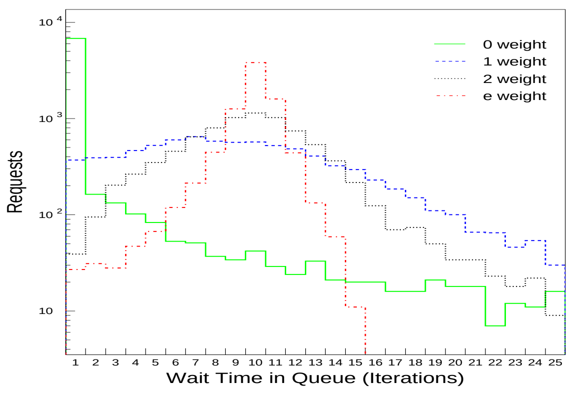


(b) Zipf Distribution.

Figure 8. Byte Miss-Ratio vs Queue Length

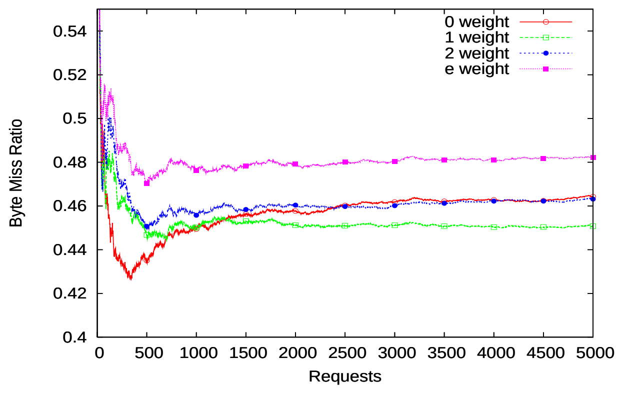


(a) Uniform Distribution.

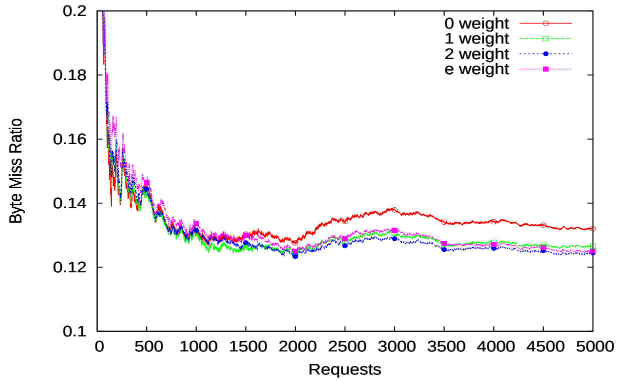


(b) Zipf Distribution.

Figure 9. Queue Wait Time (in iterations) for different “anti-starvation” functions - Log Plot



(a) Uniform Distribution.



(b) Zipf Distribution.

Figure 10. Byte Miss-Ratio for different “anti-starvation” functions

7. Conclusions and Future Work

We have identified a new type of caching problem that is notable in applications where multiple files must be in cache for an application to access them concurrently. In this case dependencies exist among files that must be cached. This problem arises in various scientific and commercial applications that use vertically partitioned attribute files and maintain each attribute in different files. Traditional cache replacement policies, where decisions as to which files should be cached or evicted, are based on one file request at a time, do not apply here since bundles of files are requested at a time. An application can only proceed if all the files requested are cached.

The problem of optimally loading the cache so as to maximize the value of satisfied requests is NP hard. We have proposed approximation algorithms that were shown analytically to produce solutions bounded from the optimal one by a factor of $1/(2d^*)$. We applied extensive simulations to compare our best proposed algorithm, *FBC* with a variant of the GreedDual-Size *GDS-MF* that handles multiple file requests at a time. The metric measure used in our comparisons was primarily the byte miss ratio. Two streams of synthetic workload that reflect the file-bundle requests of typical applications were generated. The first was based on Random request distribution and the second was based Zipf request distribution The results of our tests show that, the proposed algorithms outperform the *GDS-MF* for both distribution but the case of the Zipf distribution was significantly more pronounced. The results indicate that *FBC* involves less data transfers into and out of the cache than *GDS-MF*.

The FBC algorithm 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 FBC can be used to approximate the latter problem with the same bounds from optimality.

Although most data intensive scientific applications, involve caching of file bundles, these environments still log the files as single independent requested files. All the log traces of caching activities examined so far, present the logs as single file requests at a time. We are working with operational staff of the dCache system at Fermi Laboratory, and at NERSC to instrument the caching activities for file-bundles. These log traces will be used in future for real workload cache simulations. Future work will include the incorporation of the *FBC* algorithm in an actual application environment, such as the data-grid, where large scale data intensive scientific applications are being scheduled to run in the future. We intend to also extend this work to include cases when the processing time (duration of time to retain the file in the cache for processing) and the transfer times of files into the cache are also considered. The case of a hybrid execution model is also of interest where we have a mix of jobs some of which execute according to *One File at a Time* model while others execute according to the *File-Bundle at a Time* model.

References

[1] H. Andrade, T. Kurc, A. Sussman, E. Borovikov, and J. Saltz. On cache replacement policies for servicing mixed data intensive query workloads. In *Proc. 2nd Workshop on Caching, Coherence, and Consistency, with the 16th ACM Int'l. Conf. on Supercomputing*, New York, NY, June 2002.

[2] BaBar: The BaBar collaboration, http://www.slac.stanford.edu/BFROOT/.

[3] W. Bethel, B. Tierney, J. Lee, D. Gunter, and S. Lau. Using high-speed WANs and network data caches to enable remote and distributed visualization. In *Supercomputing*, 2000.

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

[5] M. Ernst, P. Fuhrmann, M. Gasthuber, T. Mkrtchyan, and C. Waldman. dcache: A distributed data caching system. In *Computing In High Energy And Nuclear Physics, CHEP'01*, 2001.

[6] U. Feige, D. Peleg, and G. Kortsarz. The dense k-subgraph problem. *Algorithmica*, 29(3):410–421, 2001.

[7] 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.

[8] E. J. Otoo, D. Rotem, and A. Shoshani. 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, Jun., 21 2003. IEEE Computer Society, Los Alamitos, California.

[9] B. Reiner and K. Hahn. Optimized management of large-scale datasets stored on tertiary storage systems. *IEEE Distributed Systems Online Magazine*, Mar. 2004.

[10] X. Shen and A. Alok Choudhary. A distributed multi-storage i/o system for data intensive scientific computing. *Parallel Comput.*, 29(11-12):1623–1643, 2003.

[11] A. Shoshani, A. Sim, L. M. Bernardo, and H. Nordberg. Coordinating simultaneous caching of file bundles from tertiary storage. In *Proc. 12th Int'l. Conf. on Scientific and Stat. Database Management, SSDBM'2000*, pages 196 – 206, 2000.

[12] M. Tan, M. Theys, H. Siegel, N. 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.

[13] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin, 2001.

[14] J. Wang. A survey of web caching schemes for the internet. In *ACM SIGCOMM'99*, Cambridge, Massachusetts, Aug. 1999.

[15] K. Wu, W. S. Koegler, J. Chen, and A. Shoshani. Using bitmap index for interactive exploration of large datasets. In *SSDBM'2003*, pages 65–74, Cambridge, Mass., 2003.

[16] 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.

A. Proofs of Lemmas and Theorems

A.1. Primal dual bound from optimal

The FBC 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 FBC can be stated as:

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

subject to

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

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

In this formulation the objective is to maximize the sum of the value of requests addressed using files in the cache. The first set of constraints ensure that y_j is less than all the z_i's corresponding to files used by request j. On the other hand if all the files required by r_j are in the cache the optimal solution will set y_j equal to one. The second set of constraints ensure that sum of the sizes of files in the cache does not exceed the size of the cache, $s(C)$. The linear relaxation of this problem,$\mathcal{P}_1$, and its associated dual problem, $\mathcal{D}$, are not only easier to analyze but also provide a useful bound for a heuristic solution procedure.

$$\mathcal{P}_1 : \quad \max \sum_{j=1}^{n} v(r_j) y_j$$

subject to

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

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

In this formulation we have relaxed the 0-1 restriction on the z_i's. The advantage of the relaxation is that we are able to use the dual of the problem to obtain bounds. The technique used by us to obtain the bound is discussed in detail in Vazirani [13].

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

subject to

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

$$\lambda s(f_i) + \lambda_i \quad - \sum_{j : f_i \in F(r_j)} \lambda_{ji} \geq 0 \qquad (2)$$

For $i = 1, 2, \ldots, m,\ \lambda, \lambda_i, \lambda_{ji} \geq 0$. The interpretation is as follows: 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.

We shall use the linear programming relaxation to bound the solution produced by OptCachSelect. This will be done in two steps. First we shall bound the solution to $\mathcal{P}_1$, by upper bounding the solution to D. Then we produce a feasible solution to the primal that can be compared to this bound. Then we will bound OptCachSelect. Consider an approximation algorithm OptCachSelect(LP) for solving $\mathcal{P}_1$ that is similar to OptCachSelect except that it allows partial loading of files. It comprises of ranking the requests in descending order of the $v'(r_j)$'s and loading them greedily until the cache is full. Assume that if all the files for a request cannot be fully loaded, they are loaded partially until the cache is full. Let's assume that the collection of requests serviced from the cache without loss of generality are denoted as $r_1, r_2, \ldots, r_p$. Now, we exhibit the feasible dual solution. Let

$$\lambda_{ji} = \frac{v(r_j) s(f_i)/d(f_i)}{\sum_{t \in F(r_j)} s(f_t)/d(f_t)}$$

It can be verified algebraically that this assignment to the λ_{ji}'s satisfies the constraints (1). Let

$$\lambda = \frac{v(r_p)}{\sum_{t \in F(r_p)} s(f_t)/d(f_t)} = v'(r_p).$$

Set λ_j to 0 for files not used by the p requests as well as the files used to address only the p^{th} request. Then for this assignment of dual variable values, the left hand side of (2) evaluates to

$$\begin{aligned}
\lambda s(f_i) &+ \lambda_i - \sum_{j:f_i \in F(r_j)} \lambda_{ji} \\
&= s(f_i) \frac{v(r_p)}{\sum_{t \in F(r_p)} s(f_t)/d(f_t)} \\
&- s(f_i) \sum_{j:f_i \in F(r_j)} \frac{v(r_j)/d(f_i)}{\sum_{t \in F(r_j)} s(f_t)/d(f_t)} \\
&\geq s(f_i) \left(v'(r_p) - \max_{j \geq p} v'(r_j) \right) \geq 0
\end{aligned}$$

because $v'(r_p)$ is greater than or equal to $v'(r_j)$ of any unloaded request.

Thus equation 2 are satisfied for such files. Finally, for files used to address the $p-1$ requests, let

$$\lambda_i = \max_{j<p, i \in F(r_j)} \left\{ s(f_i) \left(v'(r_j) - v'(r_p) \right) \right\}.$$

A similar substitution as above reveals that

$$\begin{aligned}
\lambda s(f_i) &+ \lambda_i - \sum_{j:f_i \in F(r_j)} \lambda_{ji} \\
&= s(f_i) \frac{v(r_p)}{\sum_{t \in F(r_p)} s(f_t)/d(f_t)} \\
&+ \max_{j<p, i \in F(r_j)} \left\{ s(f_i) \left(v'(r_j) - v'(r_p) \right) \right\} \\
&- s(f_i) \sum_{j:f_i \in F(r_j)} \frac{v(r_j)/d(f_i)}{\sum_{t \in F(r_j)} s(f_t)/d(f_t)} \\
&\geq s(f_i) \left(v'(r_p) - \max_{j<p} v'(r_j) \right) \\
&+ \max_{j<p, i \in F(r_j)} \left\{ s(f_i) \left(v'(r_j) - v'(r_p) \right) \right\} \\
&= 0
\end{aligned}$$

Finally, the dual objective function value equals

$$\begin{aligned}
& s(C)v'(r_p) \\
&+ \sum_{i \in \cup_{j<p} F(r_j)} \max_{j<p, i \in F(r_j)} \left\{ s(f_i) \left(v'(r_j) - v'(r_p) \right) \right\} \\
&\leq \left(s(C) - \sum_{i \in \cup_{j<p} F(r_j)} s(f_i) \right) v'(r_p) \\
&+ \sum_{i \in \cup_{j<p} F(r_j)} \max_{j<p, i \in F(r_j)} \left\{ s(f_i) \right\} v'(r_j) \\
&\leq \left(s(C) - \sum_{i \in \cup_{j<p} F(r_j)} s(f_i) \right) v'(r_p) \\
&+ \sum_{j<p} v'(r_j) \sum_{i \in F(r_j)} s(f_i) \\
&\leq \max_i d(f_i) \left(\sum_{j<p} v(r_j) \right. \\
&+ \left. \frac{\left(s(C) - \sum_{i \in \cup_{j<p} F(r_j)} s(f_i) \right)}{\sum_{i \in F(r_p)} s(f_i)} v(r_p) \right). \qquad (3)
\end{aligned}$$

In the second inequality we have used the fact that the maximum of a sum of positive values is less than their sum. The final expression equals the value of the solution produced by the approximation algorithm times the maximum number of requests that need the same file. However, the objective function value of any feasible solution to the dual is greater than the value of the optimal solution to primal.

Theorem A.1. *Let $V_{OptCachSelect}$ represent the value produced by Algorithm OptCachSelect 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*

$$\frac{V_{OPt}}{V_{GR}} \leq 2d^*$$

Proof. Outline: Modify the algorithm $OptCachSelect(LP)$ such that it stops with the last request that can only be accommodated partially (or not at all). It then also compares the solution produced to the value of the last request that could not be accommodated and outputs the larger of the two solutions. As one of the two terms within the parentheses on the right hand side of equation (3) is larger than the other; the integral solution produced by the modified $OptCachSelect(LP)$ is at least $1/2d^*$ times the optimal solution. Algorithm $OptCachSelect$ can be adapted to produce equivalent or a better solution then $OptCachSelect(LP)$ $\square$