Optimal File-Bundle Caching Algorithms for Data-Grids

Ekow Otoo and Doron Rotem and Alexandru Romosan
Lawrence Berkeley National Laboratory
1 Cyclotron Road, MS: 50B-3238
University of California, Berkeley, CA 94720

Abstract

The file-bundle *caching problem arises frequently in scientific applications where jobs need to process several files simultaneously. Consider a host system in a data-grid that maintains a staging disk or disk cache for servicing jobs of file requests. In this environment, a job can only be serviced if all its file requests are present in the disk cache. Files must be admitted into the cache or replaced in sets of* file-bundles*,, i.e. the set of files that must all be processed simultaneously. In this paper we show that traditional caching algorithms based on file popularity measures do not perform well in such caching environments since they are not sensitive to the inter-file dependencies and may hold in the cache non-relevant combinations of files. We present and analyze a new caching algorithm for maximizing the throughput of jobs and minimizing data replacement costs to such data-grid hosts. 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, etc. In all these cases, the results show significant improvement as compared with traditional caching algorithms.*

1 Introduction

A data-grid [2] defines an environment of the GRID model of computing that encompasses the management of large numbers of data sets that result from current scientific experiments and observations and are also likely to result in future scientific research. Specific examples of such collaboratory research activities are the Particle Physics Data Grid (PPDG) [5] and the Earth Science Grid (ESG) [4]. The general model of computing within these research communities can be characterized as:

- having globally geographically dispersed locations of experimental laboratories and/or observatories that constitute widely distributed and heterogeneous data sources;
- maintaining different models of storage resources, e.g., Mass Storage Systems (MSS) from different vendors and Storage Area Networks (SANs).
- Computing on high performance computing systems that are located at sites far removed from the sites of the researchers and data resources.

The applications in these domains, typically referred to as data intensive applications, require not only high performance computing resources, but timely high speed access to the data resources (generally maintained in units of data files), that are needed at the computational resources. To access these large heterogeneous distributed data over wide area network, there is the need to implement strategies that significantly improve the data access performance under different models of computing. Techniques for efficient and optimal data access in these environments involve implementation of good file request schedules, application of optimal file caching (or data staging), usage of strategic data replication and pre-fetching, and application of efficient cache and replica replacement algorithms.

Caching techniques, in particular, have been used generally to improve the performance of storage hierarchies in computing systems. In the data-grid environment, specialized middle-ware services, such as Storage Resource Manager (SRM) [12] and Storage Resource Brokers(SRB) [11], provide the intermediary services for caching or staging files required by data intensive jobs. An SRM provides service by holding, for some duration of time, data that are requested by multiple clients, providing a uniform consistent interface to clients making file requests to heterogeneous MSS, masking intermittent link failures and data transfer disruptions that would otherwise have to be handled by clients. The file caching activities of an SRM and cache replacement are the main issues we focus on in this paper.

A Storage Resource Manager (SRM), runs on a host, or a cluster of machines, that receives job requests submitted to a data-grid. Each file can be very large (of the order of few to tens of gigabytes), and typically resides in an MSS

that is either local or remote. An SRM maintains a large capacity disk cache, of the order of hundreds of gigabytes to tens of terabytes. The disk cache is used to retain those files that are read from or are to be written to Mass Storage Systems and is shared by multiple jobs. An SRM's host that consists of a cluster of machines may have its disk cache distributed over independent disks of the cluster nodes. The use of a storage resource manager is analogous to the use of a proxy-server and/or a reverse proxy in web-caching except that SRMs deal with very large data files. In particular, a large number of file requests may be batched in one job. As a result, SRMs contend with file accesses that incur significant long delays in accessing and processing files over wide area networks. The notion of a job being processed at an SRM involves simply migrating the data that are requested to the computational resource, possibly with some transformations applied. The transformation carried out may simply be a filtering process that restricts the data to the subset that satisfies some query condition. Similar related services are rendered by a Storage Resource Broker (SRB). In general, the operation of an SRM is governed by a set of policies such as the *the job service (or scheduling) policy*, the *file caching policy*, and the *cache replacement policy.*

1.1 Problem Definition and Motivating Examples

We address the problem of cache replacement for disk caches, where replacements occur in file-bundles, in the context of an SRM. The result is equally applicable to staging disks of mass storage systems and storage area networks (SAN). Consider a sequence of jobs that make requests for files at an SRM where each job is comprised of 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 served only if all the files that it needs are already in the cache C, otherwise the requested files of the job must be retrieved from a Mass Storage System, located either locally or at a remote site, and at a much higher cost in time, into C. The problem being addressed here is the following. Given that each job requires all its requested files to be in cache before service begins, what is an optimal cache replacement policy to achieve the maximum throughput, or alternatively minimize the volume of data transfers, under a limited cache space.

There are many papers [1, 6, 8, 13, 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 files requested but not found in the cache. The assumption used by these works is that each request is associated with exactly one file. The problem discussed in this paper is different and more general than these earlier works. In our case each arriving request may need to load multiple files simultaneously into the cache rather then one file at a time.

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 m-way join requires that m files must be in cache at the same time making it necessary to make cache loading and replacement decisions based on file-bundles rather then a single file at a time.

One example of an application where this problem arises is High Energy and Nuclear Physics (HENP) data analysis. The data sources are from experiments that consist of accelerating subatomic particles to nearly the speed of light and forcing their collision. A small part of the particles collide head on and produce a large number of sub-particles. Each such collision (called an event) has multiple attributes such as the total energy of the event, momentum, number of particles of each type etc. Typically the values for attributes are stored in separate files where each file stores values of an attribute across multiple events. These files are subsequently analyzed by physicists who may wish to look at several attributes simultaneously in order to select some subset of "interesting events.

Another example of this situation is simulation programs of climate modeling. These programs 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 (see Fig. 1).

A third example of simultaneous retrieval of multiple files comes from the area of bit-sliced 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 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.

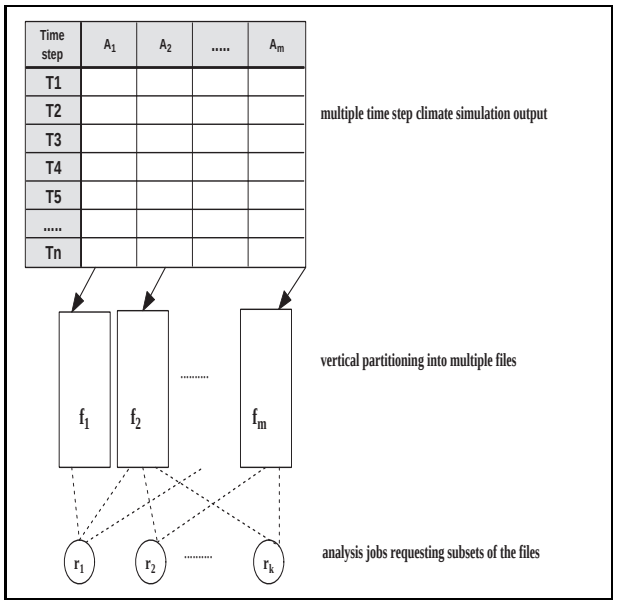


Figure 1. Illustration of vertical partitioning of climate simulation data

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.

The solution to the problem discussed in this paper serves as a fundamental building block in the design of efficient cache service policies. Given an available space in the cache and a collection of requests currently waiting in the admission queue, the goal is to load files into the available space in the cache to maximize the number of requests that can be serviced and consequently minimize the average response time of the jobs. Alternatively, since the cost (in time) to retrieve files into the cache is unusually high, we can solve to minimize the average volume of data read into the cache per request for a given workload trace.

1.2 Performance Metrics

Cache replacement algorithms are key to the implementation of a good caching system. Not only should this be evaluated in an almost negligible time relative to the time it takes to cache an object, but it should optimize, in some sense, some measure of a performance metric. 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*).

Consider a workload of a sequence of N jobs $R = \langle r_1, r_2, \ldots r_N \rangle$, where each job $r_i = \{f_i\}$ makes a request for only one file f_i . We denote the size of the file f_i by $s(f_i)$. 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 $1 - \rho_{byte-hit}$.

Numerous techniques have been proposed for optimal cache replacement all of which are geared towards either maximizing the *hit ratio* or the *byte hit ratio*. These techniques generally retain in the cache either the most frequently referenced objects or the most recently referenced objects. The former effectively evicts the least frequently used object (i.e., the LFU-policy), the latter evicts the least recently used object (i.e., the LRU-policy). Algorithms in web-caching try to minimize the *byte miss ratio* [1, 14].

Our algorithms are 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*. 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 impacted when queues of waiting jobs are taken into consideration.

1.3 Main Results

The main results of this paper are:

1. Identification of a new caching problem, that arises frequently in scientific applications that deal with vertically partitioned files.

2. Derivation of a new cache replacement algorithm (*OptFileBundle*), 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.

3. Results of extensive simulation runs that compare the *OptFileBundle* algorithm with *Landlord* [16] cache replacement consistently show that *OptFileBundle* gives a much lower average volume of data transfers per request with file requests observing either Uniform or Zipf distributions.

4. The heuristic algorithm *OptCacheSelect* used by *OptFileBundle* 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 $(1-e^{-1/d})$ from 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 present the application environment in the context of a data-grid. In Section 3, a greedy heuristic algorithm, called *OptFileBundle* is presented. The theoretical foundation of the *OptFileBundle* algorithm is presented in Section 4 where characteristics of the algorithm, e.g., its bounds from the optimal solution, are discussed. In Section 5, a simulation of the *File-Bundle Caching* is presented using the *OptFileBundle* caching algorithm. The performance of our proposed algorithm is compared with one of the best performing existing caching algorithms that does not use file-bundles in its replacement decisions. The results of the experiments carried out are also discussed in this section. We conclude in Section 6 with summary and directions for future work.

2 Data-Grid and Storage Resource Managers

Distributed scientific applications anticipated in the next several years would require access to large amounts of data of the order of hundreds of terabytes to tens of petabytes. The envisioned model of managing and accessing the data is through what is currently referred to as data grids where the data repositories are maintained in mass storage systems and are accessed from different locations by large communities of scientists. The term data-grid generally imply any distributed network infrastructure of storage resources and repositories of huge amounts of data coming from scientific experiments in the following disciplines: High Energy Physics, Biology, Earth Observation Systems and Astrophysics. The idea is to support scientific explorations that require intensive computations and analyses of large-scale shared databases across widely distributed scientific communities.

The role of an SRM is depicted in Figure 2. Its primary function is both as a proxy server and a reverse proxy server according to their specialized usage. For example a disk resource manager (DRM), manages disk resource only while a hierarchical resource manage (HRM) manages data flow into and out of Mass Storage Systems. In either case, it attempts to deliver, in a timely manner to a computational resource, the data being requested. The file requests are batched in the form of jobs and the service rendered is prescribed by either the job or the policy of the SRM. There are different models for servicing jobs.

One File at a Time: This model of service requires that, for each job, only one of the number of possible files being requested needs to be serviced at a time and this could be in some defined order or in any order.

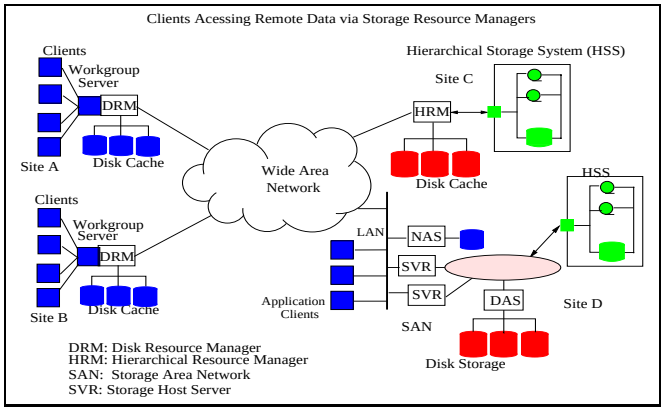


Figure 2. An SRM in the Context of a Data-Grid

One *File-Bundle* at a Time: The term *File-Bundle* refers to the set of files that a job expects to be available in cache for it to run. The job may actually by considered as composed of tasks (or sub-jobs), where each task requests a file-bundle and the job completes after executing all its tasks. The number of files in a bundle, i.e., the *file-bundle* size, varies from bundle to bundle.

An optimal service policy for the first model of service was addressed in [8]. We consider here the *File-Bundle at a Time* when jobs are percieved as a sequence of independent tasks. By *optimal service* we mean one that either maximizes the throughput of jobs or minimizes the average volume of data replaced in the cache per request.

2.1 Related Work

Storage resource managers and component prototypes are already in service [12, 11]. Techniques for transferring large data files and database objects over wide area networks have been the subject of extensive research studies for many years in distributed and federated databases [10]. One major idea resulting from these early works is that improved response times are achieved with extensive data replication, caching and data staging [13, 9] on proxy servers. A close analogous environment from which one could draw some experience and relate it to the *file-bundle* problem is in web-caching. For example, web-caching [1, 14], address similar cache replacement policies except that the scale of data sizes and transfer delays considered are on a much smaller scale than those in a data-grid environment.

3 A Caching Algorithm Based on File Bundles

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

Example: Let us assume that we have six possible requests $r_1, r_2, ..., r_6$ each associated with one or more files from $F = f_1, f_2, ... f_7$ as shown by the lines connecting requests to their associated files in Fig. 3. 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. probabilty of $\frac{1}{6}$ that any of the requests is the next one to arrive. Each row in Table 1 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 2 shows request-hit probabilities,i.e., the probablity that a random request will find all the files it needs 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.

The previous example illustrates the need for caching strategies that take into account request-hits rather than simple file-hit based algorithms. We will now proceed to describe our caching algorithm.

At the heart of our caching strategy is an algorithm called *OptCacheSelect* that determines the files that must be replaced. It takes into account file sizes and request type distributions and will be described in more detail below. The result produced by *OptCacheSelect* is a new set of files loaded into the cache that attempt 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.

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 1. File request probabilities

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 2. Request-hit probabilities

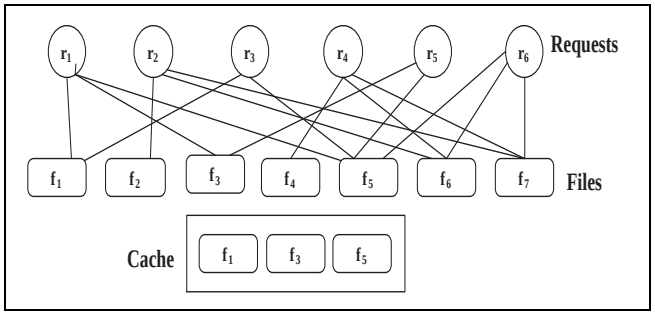


Figure 3. Example of file selection

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 all 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.

- 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 served, 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.

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 incur any additional cache resources. This leads to an increase in adjusted value for requests that share files with the last selected request.

We are now in a position to describe the main steps of our caching algorithm, *OptFileBundle*, as illustrated in Fig. 4. 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. 4a). At some point the cache fills up (Fig. 4b) and a caching replacement decision must be taken when a new request, r_{new}, arrives.

All files requested by r_{new} that are not currently present in the cache must be must loaded into the cache and some other files currently in the cache must be evicted in order to make space for them (Fig. 4c). 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. 4d).

The algorithm *OptFileBundle* works as follows:

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 $r_1, \ldots, r_n$ 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

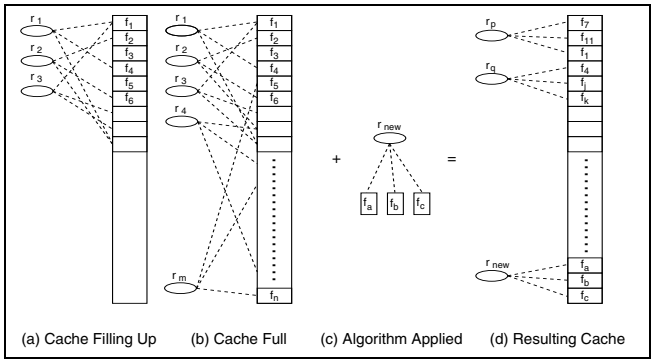


Figure 4. The steps of algorithm OptFileBundle

4 Complexity Analysis of the Algorithms

As *OptCacheSelect* is attempting to maximize request-hit probability and the value of each request approximates its populatity it follows that the fundamental problem that

input : A new request r_{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_{new}

Algorithm 2: Algorithm OptFileBundle

the *OptCacheSelect* algorithm is trying to solve can be presented 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 $s(C)$, 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 $s(C)$.

We will call this the File-Bundle Caching (FBC) problem. It is easy to show that in the special case that each file is needed by exactly one request the FBC problem is equivalent to the well-known knapsack problem. The cache C is the knapsack and its size, $s(C)$ represents the knapsack capacity, each request corresponds to an item of value $v(r_i)$ and weight equal to the total size of the files needed by r_i, i.e., size of $F(r_i)$. It is more interesting to note that the FBC problem is NP-hard even for the restricted case that each request has exactly 2 files of equal size. This is done by reduction from the Dense $k-$subgraph (DKS) problem [3]. 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 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 [3] is within a factor of $O(|V|^{1/3-\varepsilon})$ from optimum for some $\varepsilon > 0$. It is also conjectured in [3] that an approximation to DKS with a factor of $(1+\varepsilon)$ is NP-hard. Let us define d as the maximum number of requests that need the same file. For example in the system described in Fig. 3 $d = 4$ as f_5 is used by 4 different requests. In Appendix A we prove that the total value of the requests loaded by *OptCacheSelect* is within a factor of at most $\frac{1}{2}(1-e^{-1/d})$ from the value of an optimal solution. This can be stated as:

Theorem 4.1. *The total value of requests supported by the cache as loaded by algorithm* OptCacheSelect *is within a factor of* $\frac{1}{2}(1-e^{-1/d})$ *of the optimal value of* $v(OPT)$.

We can show that methods similar to the ones used in [7] can improve this bound by a factor of 2 to $(1-e^{-1/d})$ at higher computational cost. This is done by constructing a candidate solution consisting of selecting k requests for some small fixed k ($k = 2$ is a valid choice) and complementing it by running *OptCacheSelect* on the remaining space in the cache and remaining requests. Iterating this procedure over all possible choices of k requests and selecting the solution with highest value among all these candidate solutions results in the above improved bound.

5 The Simulation Model and its Environment

We designed a simulation model to explore how the *OptFileBundle* algorithm compares with the *Landlord* algorithm [1, 16]. For that purpose, we implemented a version of *Landlord* where each job makes a request for a set of files rather than a single file as in the original implementation. The implementation is described below in Algorithm 3. The simulation program, *cacheSim*, was written in $C++$ with extensive use of STL. Using a cluster of two 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 for different workload distributions, and varying cache sizes. We note here that we mesaure cache sizes by the number of requests that can be accommodated in the cache. This is a slight departure from the general method of reporting results in which the absolute cache size is specified.

input : A new request, r_{new}, a cache C of size $s(C)$, $F(C)$. the set of files currently in cache.

output : The solution G - a subset of the files in $F(C)$ that must be evicted from the cache to make room for r_{new}.

Step 0: Maintain a value credit[f] $\in [0,1]$ with each item f in the cache.
Step 1: Compute S, the amount of space needed by files in $F(r_{\text{new}})$ that are not currently in the cache C.
Step 2: Compute $F(C') = F(C) \backslash F(r_{new})$, the subset of files in cache C which are not requested by r_{new}.
Step 3:
while $s(r_{\text{new}}) > s(C) - s(F(C))$ **do**
  for $f \in F(C')$ **do**
    Find the minimum credit;
  end
  Decrease all credits by the minimum credit;
  Evict from the cache the subset of items f such that credit[f] $= 0$;
end
Step 4: Bring $F(r_{\text{new}})$ into the cache and $\forall g \in F(r_{\text{new}})$ set credit[g] $\leftarrow 1$;

Algorithm 3: Algorithm Landlord

5.1 Workload Characterization

Although file-bundle is the mode of file request in most data intensive scientific analysis, efforts have not be made to derive workload traces and logs of caching activities. The log traces maintained by most scientific centers are mainly for one file at a time requests. Even in web-caching environments, the log traces are still on per file basis.

In the absence of runtime measured workload characteristics, we constructed a simulated workload consisting of a given set of jobs, with each job requesting a random number of files from the 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. Given a defined cache size, the size of each file was generated randomly between a minimum size of 1MB and a maximum size expressed as a percentage of defined cache size that varied from 1% to 10%. The set of files requested by each job was chosen randomly from the list of available files such that the total size of the files requested was smaller than the available cache size. Each simulation run consists of submitting a number of jobs (typically 10000) in order to study the effects of the various parameters.

5.2 Simulation Parameters

There are several parameters that can be varied to observe different affects on the performance measures. We describe some of the major ones varied in our simulations:

Request Size: This refers to the total size of files needed by a request. Assuming a cache of a fixed size, the average request size determines the number of requests that can be supported in the cache at any given time. The more requests that are already in cache, the more likely that files requested by an incoming job are already present in cache. Conversely, assuming a fixed request our results translate to studying the effect of varying the cache size.

Popularity Distribution: The popularity distribution of requests for typical workloads is hard to characterize in the case of file bundles since each request draws a random combination of files and two requests are identical only if their file requests are the same. As a result we examine the effects of the two extreme distributions: a purely random distribution, and a Zipf distribution.

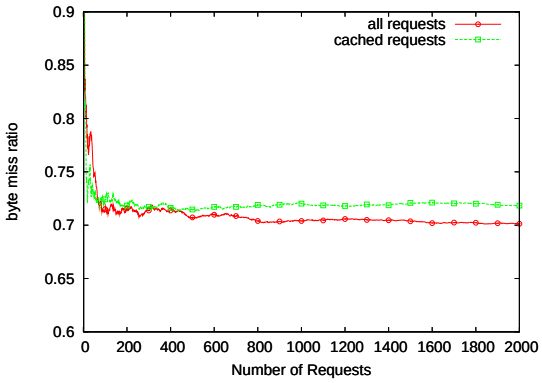
Incoming Queue Length: Instead of processing a job in first come first serve order, one can also consider aggregating the jobs in an admission queue of a given length, and admit the next job to be processed according to some fair effective scheduling algorithm, i.e., one that avoids *request lockout* but at the same time minimizes the *byte miss ratio*

Request History Length: As shown in this paper, the *OptFileBundle* algorithm determines the optimal requests to be loaded based on the history of all requests ever made. Computationally this gets to be expensive as the number of requests increases. We studied the effect of truncating the history length to consider only the requests supported by the cache while obtaining the request popularity and the degree of file sharing from the global history.

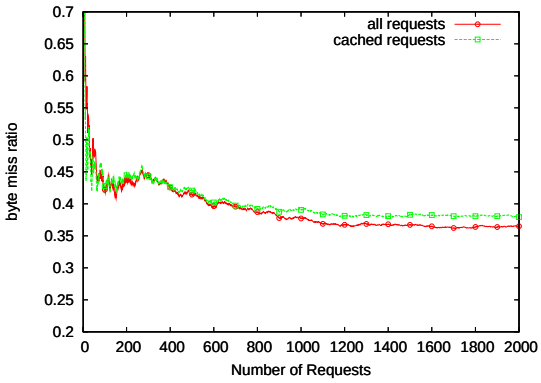
5.3 Simulation Results

We present some results of simulation runs with variability in some representative parameters such as varying cache size and different request distributions.

Computational costs for each iteration are a function of many factors, but depend strongly on the average number of requests served by the cache, and for the $OptFileBundle$ algorithm by the number of requests in the history. As the number of requests increases, the computational costs become expensive. As such, we studied the effect of truncating the history length. A variety of approaches were explored, from arbitrarily limiting the history to the requests

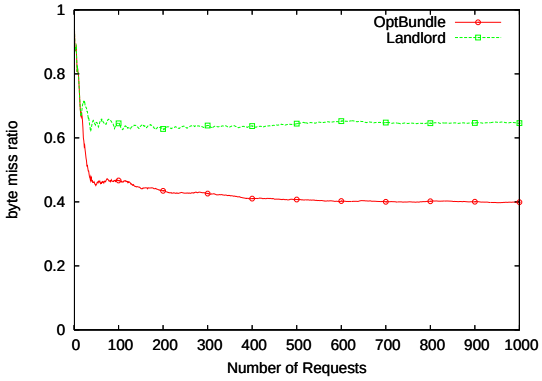


(a) Random distribution.

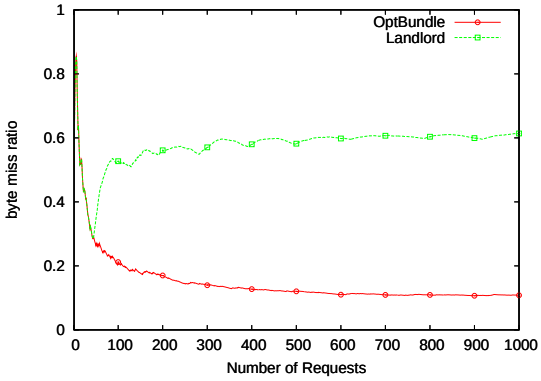


(b) Zipf distribution.

Figure 5. Effect of varying the history length

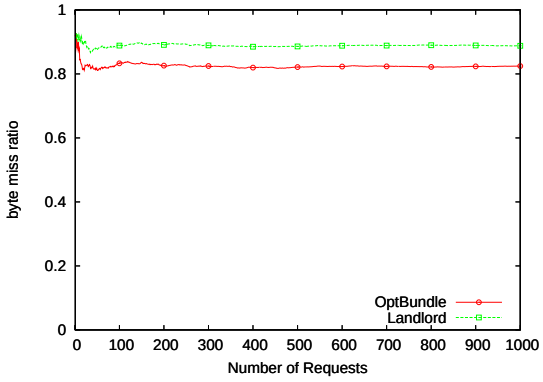


(a) Random distribution.

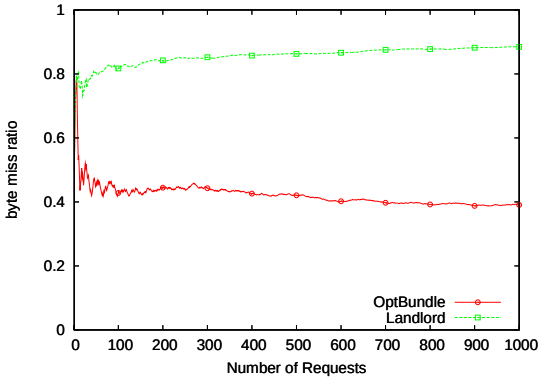


(b) Zipf distribution.

Figure 6. Byte miss-rate for small files.

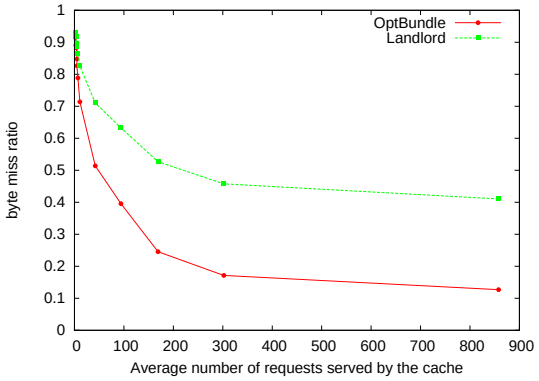


(a) Random distribution.

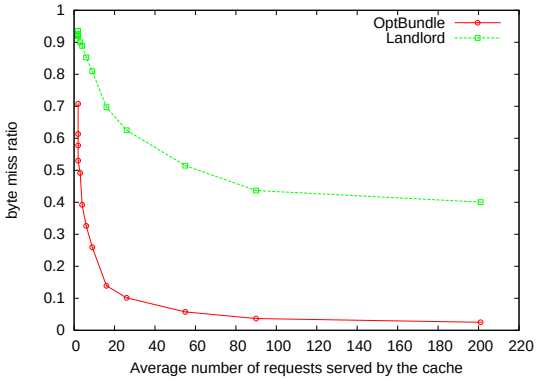


(b) Zipf distribution.

Figure 7. Byte miss-rate for large files.

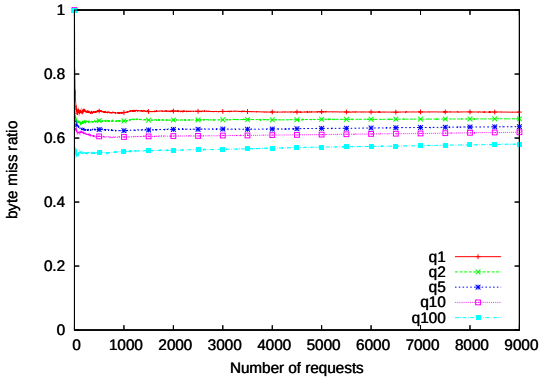


(a) Random distribution.

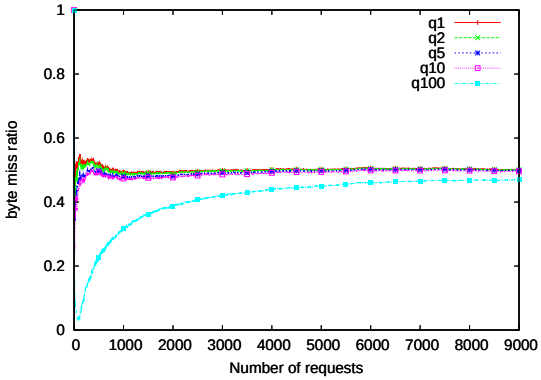


(b) Zipf distribution.

Figure 8. Effect of varying the cache size



(a) Random distribution.



(b) Zipf distribution.

Figure 9. Effect of varying the queue length

in the cache to a full history of all requests ever processed are retained. As shown in Fig. 5, the effects of such truncation are negligible, and subsequent simulations were run using only the truncated history limited to the requests in the cache. This approach has the advantage that, for a given set of parameters, computational costs stay constant per iteration.

The first set of experiments performed involved job popularity distributions. A random (or 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. In addition, we varied the size of the incoming requests, implicitly varying the size of the cache.

Figures 6(a) and 6(b) compare the *byte miss ratio* for random and Zipf request distributions. In both cases the cache replacement based on *OptFileBundle* algorithm performs better than the *Landlord* algorithm in the sense that the byte miss ratio is much lower. The superiority of *OptFileBundle* over *Landlord* is even more significant for smaller file sizes as illustrated in Figures 6(a) and 6(b). In the graphs of both Figures 6 and 7, we also observe that the values of the *byte miss ratio* are much lower for Zipf's distribution of requests than for random distributions.

The overall effect of varying the request size (and implicitly the cache size) is shown in Fig. 8. As the cache is able to serve more requests the amount of data moved into the cache for each request is smaller. This difference in the amount of data moved into the cache, between *OptFileBundle* cache replacement and *Landlord*, is even more pronounced for Zipf request distribution.

Another set of experiments we performed involved aggregating the requests in a processing queue of varying length instead of processing each request in first come first serve (FCFS) order. Once the queue is full, we first serve the request of highest relative value in the queue using $OptFileBundle$ and repeat this process on the remaining requests in the queue until it becomes empty. Fig. 9(a) and Fig. 9(b) show the effects of varying the processing queue length from 1 to 100 (shown as q1,q5,..,q100) for incoming random and Zipf request distributions, respec-

tively. The effect of queuing incoming requests is minor for uniform request distribution and the small increase in efficiency doesn't justify the additional computational costs in this case. However, the effect of queuing is more significant for Zipf request distribution as a queue of size 100 creates a much lower *byte miss ratio* as compared with smaller queue sizes.

6 Conclusion and Future Work

We have presented the *file-bundle caching* problem that arises frequently in scientific applications where jobs need to process several files simultaneously. Unlike traditional approaches to disk cache replacement where one file is requested at a time, this problem requires that a set of files be cached simultaneously at a time. Given such a request stream, and a defined cache size, the question is which set of files in the cache are to be replaced so that the *miss byte ratio* is minimized. We have presented a greedy heuristic algorithm, *OptFileBundle*, that determines the set of files that are evicted at each request admission. In particular we showed that at each instance of a request arrival, the set of files replaced by the algorithm, gives a solution that is within a factor of $(1 - e^{-1/d})$ of the optimum, where d is the maximum number of requests per file, and e is the base of the natural log. The solution has particular significance in establishing policies for Storage Resource Managers and similar software that service file requests, e.g., Storage Area Networks SANs and other storage request brokers in data intensive grid environments.

We also presented simulation results that support the efficiency of our cache replacement algorithms. The *Landlord* cache replacement algorithm is one of best known for disk file caching. Using the *miss byte ratio* as the performance metric, we compared the *OptFileBundle* and *Landlord* under varying simulated workloads of request streams and cache sizes. For both uniform requests streams, i.e., one in which every request is equally likely, and Zipf request distributions we find that *OptFileBundle* consistently gives a lower value of *miss byte ratio* than the *Landlord*. However, the values of the *byte miss ratio* are much lower for Zipf's distribution of requests than for random distributions.

Future work will address the incorporation of *OptFileBundle* algorithm in an actual Storage Resource Manager systems currently under development at our lab. 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.

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

[2] 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.

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

[4] Earth Science Grid. http://www.scd.ucar.edu/css/esg/.

[5] Particle Physics Data Grid. http://www.ppdg.net/.

[6] 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.

[7] S. Khuller, A. Moss, and J. S. Naor. The budgeted maximum coverage problem. *Information Processing Letters*, 70(1):39–45, 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] E. J. Otoo and A. Shoshani. Accurate modeling of cache replacement policies in a data grid. In *11th NASA Goddard Conf. on Mass Storage Syst. and Tech. / 20th IEEE Symp. on Mass Storage Syst.*, San Diego, California, April 7 - 10 2003.

[10] T. T. Ozsu and Valduriez. *Principles of distributed database systems*. Prentice Hall, Upper Saddle River, N.J., 2nd edition, 1999.

[11] A. Rajasekar, M. Wan, and R. Moore. Mysrb & srb - components of a data grid. In *The 11th Int'l. Symp. on High Perf. Distrib. Comput. (HPDC-11)*, Edinburgh, Scotland, Jul. 24 - 26 2002.

[12] A. Shoshani, A. Sim, and J. Gu. Storage resource managers: Middleware components for grid storage. In *10th NASA Goddard Conference on Mass Storage Syst. and Tech.*, Apr. 15 - 18 2002.

[13] 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.

[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

For a set of requests G, let $v(G)$ denote the total value, i.e., $v(G) = \sum_{r \in G} v(r)$. Let OPT denote the set of requests selected by the optimal solution. Let r denote the number of iterations made by algorithm *OptCacheSelect*, until a request from OPT is rejected because of insufficient space in the cache. Assuming that during the first r iterations, l requests are added to G, we can renumber the requests $\{r_1, r_2, \ldots r_l, r_{l+1}\}$ so that r_i is the i^{th} request added to G and request r_{l+1} is in OPT but rejected by *OptCacheSelect* due to insuffient space in the cache. Let j_i denote the iteration when request r_i is considered

Lemma A.1.

$$\sum_{r_m \in OPT} s'(F_m) \leq s(C).$$

Proof. The Lemma claims that the total adjusted size of files loaded by OPT into the cache is smaller than the the cache size. Let C_{opt} denote the the set of files loaded into the cache by OPT. Then

$$\sum_{f_j \in C_{opt}} s(F_j) \leq s(C).$$

On the other hand

$$\begin{aligned} \sum_{r_m \in OPT} &= \sum_{\substack{f_j \in F_m \\ r_m \in OPT}} s'(F_j) \leq \sum_{f_j \in C_{opt}} d(f_j) s'(f_j) \\ &= \sum_{f_j \in C_{opt}} d(f_j) \frac{s(f_j)}{d(f_j)} = \sum_{f_j \in C_{opt}} s(f_j). \end{aligned}$$

The first inequality follows from the fact each file f_j in the cache is accounted for at most $d(f_j)$ times in the first two sums. The second inequality follows directly from the definition of $s'(f_j)$. □

Lemma A.2. *After each iteration $j_i, i = 1, 2, \ldots, l+1$,*

$$v(G) - v(G_{i-1} \geq \frac{s'(F_i)}{s(C)}(v(OPT) - v(G_{i-1})). \quad (1)$$

Proof. After iteration j_{i-1}, the maximum adjusted value among all requests in $OPT \backslash G_{i-1}$ (i.e., requests in OPT but not in G_{i-1}) is at most $\frac{v(r_i}{s'(F_i)}$, i.e.,

$$\begin{aligned} v(OPT) - v(G_{i-1}) &= \sum_{\substack{r_m \in OPT \\ r_m \notin G_{i-1}}} v(r_m) \\ &\leq \frac{v(r_i)}{s'(F_i)} \sum_{\substack{r_m \in OPT \\ r_m \notin G_{i-1}}} s'(F_m) \leq \frac{v(r_i)}{s'(F_i)} s(C). \end{aligned}$$

The last inequality follows from Lemma A.1. Using $v(r_i) = v(G_i - v(G_{i-1})$ and substituting in 1, the result follows. □

Lemma A.3. *We have that $v(G_{l+1}$ always satisfies the following condition*

$$v(G_{l+1}) \geq \left[1 - \prod_{k=1}^{l+1} \left(1 - \frac{s'(F_k)}{s(C)}\right)\right] v(OPT).$$

Proof. The proof is by induction. We need to show that

$$v(G_1) \geq \left[1 - \left(1 - \frac{s'(F_1)}{s(C)}\right)\right] v(OPT) = \frac{s'(F_1}{s(C)} v(OPT). \quad (2)$$

The algorithm *OptCacheSelect* selects the request with the maximum adjusted value over all requests, i.e.,

$$\frac{v(r_1)}{s'(F_1)} \geq \frac{v(r_m)}{s'(F_m)}, \forall r_m \in OPT.$$

It follows that

$$\frac{v(r_1)}{s'(F_1)} \geq \sum_{r_m \in OPT} \frac{v(r_m)}{s'(F_m)} \geq \frac{v(OPT)}{s(C)}. \quad (3)$$

The first inequality of (3) uses the fact that for $a_i, b_i > 0$, if $a/b \geq a_i/b_i$ for $1 \leq i \leq n$, then $a/b \geq \sum_{i=1}^n a_i / \sum_{i=1}^n b_i$. The second uses Lemma A.1 and the definition of $v(OPT) = \sum_{r_m \in OPT} v(r_m)$. Equation 2 follows since by definition $v(G_1) = v(r_1)$. For $i > 1$,

$$v(G_i) = v(G_{i-1}) + (v(G_i) - v(G_{i-1})). \quad (4)$$

Using Lemma A.2 on the right side of equation 4, we get

$$\begin{aligned} v(G_i) &\geq v(G_{i-1}) + \frac{s'(F_i)}{s(C)}(v(OPT) - v(G_{i-1})) \\ &= \left(1 - \frac{s'(F_i)}{s(C)}\right) v(G_{i-1}) + \frac{s'(F_i)}{s(C)} v(OPT). \end{aligned}$$

Applying the induction hypothesis to $v(G_{i-1})$ in the rightmost expression of 5 and rearranging we get

$$v(G_i) \geq \left[1 - \prod_{k=1}^{i}\left(1 - \frac{s'(F_k)}{s(C)}\right)\right] v(OPT);$$

as required. □

Lemma A.4. *Let d be the maximum degree of a file F, then*

$$d\sum_{i=1}^{l+1} s'(F_i) \geq s(C).$$

Proof. The request r_{l+1} was rejected because there was not enough space in the cache C. Hence we

$$d\sum_{i=1}^{l+1} s'(F_i) = \sum_{i=1}^{l+1} d \sum_{f_j \in F_i} \frac{s(f_j)}{d(f_j)} \geq \sum_{i=1}^{l+1} \sum_{f_j \in F_i} s(f_j) \geq s(C).$$

□

Lemma A.5. *$v(G_{l+1}$ satisfies*

$$v(G_{l+1}) \geq (1 - e^{1/d})v(OPT).$$

Proof. Using Lemma A.3 and Lemma A.4, we get

$$\begin{aligned} v(G_{l+1}) &\geq \left[1 - \prod_{k=1}^{l+1}\left(1 - \frac{s'(F_k)}{s(C)}\right)\right] v(OPT) \\ &\geq \left[1 - \prod_{k=1}^{l+1}\left(1 - \frac{s'(F_k)}{d\sum_{i=1}^{l+1} s'(F_i)}\right)\right] v(OPT) \end{aligned}$$

Using the fact that $\prod_{k=1}^{l+1}(1 - \frac{s'(F_k)}{d\sum_{i=1}^{l+1} s'(F_i)})$ is maximized when all $s'(F_k)$ are equal, i.e.,

$$s'(F_k) = \frac{\sum_{i=1}^{l+1}}{l+1}, \forall 1 \leq k \leq l+1;$$

we get

$$\begin{aligned} v(G_{l+1}) &\geq \left[1 - \prod_{k=1}^{l+1}\left(1 - \frac{1}{d(l+1)}\right)\right] v(OPT) \\ &\geq 1 - \left(1 - \frac{1}{d(l+1)}\right)^{l+1} v(OPT) \\ &\geq (1 - e^{1/d})v(OPT). \end{aligned}$$

□

Theorem 1. *The total value of requests supported by the cache as loaded by algorithm* OptCacheSelect *is within a factor of $(1 - e^{-1/d})$ of the optimal value of $v(OPT)$.*

Proof. Let the maximum value request be r_t, with value $v(r_t)$. Then $v(r_t) \geq v(r_{l+1})$. Consequently,

$$v(G_{l+1}) + v(r_t) \geq v(G_l) + v(r_{l+1}) = v(G_{l+1}) \geq (1 - e^{-1/d})v(OPT).$$

Algorithm *OptCacheSelect* produces a solution that is larger than the maximum between $v(G_l)$ and $v(r_t)$ (See Step 3 of the Algorithm *OptCacheSelect*). So the value of this solution is at least $1/2(1 - e^{-1/d})v(OPT)$. □