Impact of Admission and Cache Replacement Policies on Response Times of Jobs on Data Grids

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

Abstract

Caching techniques have been used widely to improve the performance gaps of storage hierarchies in computing systems. Little is known about the impact of policies on the response times of jobs that access and process very large files in data grids particularly when data and computations on the data have to be co-located on the same host. In data intensive applications that access large data files over wide area network environment, such as data-grids, the combination of policies for job servicing (or scheduling), caching and cache replacement can significantly impact the performance of grid jobs. We present some preliminary results of a simulation study that combines an admission policy with a cache replacement policy when servicing jobs submitted to a storage resource manager. The results show that, in comparison to a first come first serve policy, the response times of jobs are significantly improved, for practical limits of disk cache sizes, when the jobs that are back-logged to access the same files are taken into consideration in scheduling the next file to be retrieved into the disk cache. Not only are the response times of jobs improved, but also the metric measures for caching policies, such as the hit ratio and the average cost per retrieval, are improved irrespective of the cache replacement policy.

1 Introduction

Communities of research scientists are increasingly using data grids [6, 4] as the environment for managing the massive datasets that result from scientific experiments and observations. Examples of such communities include collaborators in the Particle Physics Data Grid (PPDG) [8], the Grid Physics Network (GriPhyN) [5], the Earth Science Grid (ESG) [7], and a host of others. In data intensive applications that subsequently access a large number of very large data files over these wide area network, there is the need to implement strategies that significantly improve the data access performance under different workloads. Techniques for enhancing and optimizing data accesses concern good file request scheduling, caching (or data staging), data replication, cache and replica replacement. Caching techniques, in particular, have been used generally to improve the performance of storage hierarchies in computing systems. In the grid environment, specialized middle-ware services such as storage resource manager(SRM) [16] and storage resource brokers(SRB) [14], provide the intermediary services of caching or staging files required by jobs.

A storage resource manager runs on a host, or a cluster of machines, that receives jobs requests submitted to a data grid. The job requests generally are for a large number of files. Each file can be very large (of the order of a few to tens of gigabyte), and typically reside either in some mass storage or in some tertiary storage system. Consequently an SRM maintains a large capacity disk cache, of the order of hundreds of gigabyte to a terabyte, for retaining those files that are retrieved into its disk cache so that the same file can be shared by multiple jobs. For an SRM host that consists of a cluster of machines the disk cache may be 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 significantly long delays in accessing and processing files over wide area networks.

The general problem posed, from the perspective of data management, ignoring fault tolerant issues, concerns the development of a suite of policies and their combinations thereof, that optimize the transparent access to multi-terabyte or even multi-petabyte of distributed datasets in data-intensive grid computing. Our concern here is solely

from the view point of the service rendered by a host of a storage resource manager, storage resource broker (SRB), or even a storage area network (SAN). In the sequel we would refer only to SRMs although the results are equally applicable to SRBs and SANs.

1.1 Job Servicing Model of an SRM

A job that arrives at an SRM's host makes requests for hundreds or thousands of files that it processes either one at a time or in groups referred to as *"file bundles."* The one-at-time processing may be done either in some order or arbitrarily. By processing, we mean an execution of a task such as an analysis program that takes its input data from the files stored in the local disk cache. The task execution may simply involve transferring either the entire file or a subset of the file (e.g., the result of query), to the originating host of the job. A *"file bundle"* is a set of files that must all be in cache to be processed at the same time.

In general an SRM queues the jobs and later makes decisions as to which job needs to be serviced next and which file, from the batch of files of the selected job, must be retrieved into or transfered from the disk cache. If a requested file happens to be in the cache, the SRM may choose to retain it (i.e., *"pin"* the file) in its cache until after the job that requested it releases it, in which case the file is *"unpinned."* The decision of selecting the next job to be processed is governed by a rule termed *"the service policy."* The decision of which file to retrieve into the disk cache is governed by rule termed the *"file caching policy."* When a decision is made to cache a file it may have to determine which of the files currently in the cache must be evicted to create space for the incoming one. This latter decision is also governed by the rule termed the *"cache replacement policy."* The *"service policy"* and *"caching policy"* are often combined and referred to as the *"admission policy."* Other related rules, or policies may define a limit on the number of file requests that may be processed concurrently for the same job or the amount of cache space that is allowed to be taken up by files of the same job.

Research studies conducted so far on grid job execution have only addressed job scheduling (or task assignment) problem with the view to balance the load over available distributed resources in a manner that satisfies the resource requirements of each job [1, 15]. The problem is well known to be analytically intractable and consequently, studies conducted so far have been predominantly with simulations [1, 3, 10, 15]. We follow similar methodology in our work. We concern ourselves only with the impact of the interactions of job admission policies and disk cache replacement policies on job response times. We restrict the problem to a simplified model of job execution where the order of processing of the files of a job is immaterial. When a file is cached, all jobs in the queue, at the time the selection criteria was evaluated, are immediately allowed to process the file. A job can process concurrently, as many files as it finds in the cache.

1.2 The File Admission and Caching Replacement Problem

Consider job arrivals to a queue Q_a of an SRM, where each job J_i, makes a request for a set of files to be processed independently of one another, $J_i = \{f_{i,1}, f_{i,2}, \ldots, f_{i,l}\}$. The storage resource manager, maintains a disk cache of capacity C and retrieves into its cache a file f_c according to some admission policy. The decision to cache f_c is based on the content of Q_a at the time a decision is made.

When a file f_c is read into the disk cache, all the jobs that require the file at the time the selection criterion was evaluated, invoke tasks to process the file. A task denoted by $T_{i,j}$ is identified by a combined key of the job identifier J_i and the requested file identifierf_j. A job is removed from the queue Q_a only after all its file requests have been serviced.

Assuming that the jobs $J_1, J_2, \ldots, J_M$ are currently in the queue Q_a and the cache C contains the files $f_1, f_2, \ldots, f_N$. An admission policy gives the order of servicing jobs in Q_a, which in turn implies a decision as to which files must be loaded next into the cache. A naive admission policy simply serves the jobs in the order of arrival in the queue, choosing arbitrarily, a file from the set of pending file requests of the job at the front of the arrival queue. The problem is to find an admission policy that minimizes the average response time of jobs subject to some fairness criteria or user specified constraints. Possible fairness criteria include defining:

- the maximum time a request can wait in the queue before all its file requests are completely serviced.
- the maximum idle time for a job. A job is considered to be idle if it is not processing any of its requested files.
- the maximum number of files each jobs is allowed to process concurrently.

The fairness criteria may be quantified, at any instant in time, by some value assignment $v(J_i)$ for each job J_i. The assigned value may be perceived also as a time varying priority value for the job.

The metrics for measuring the performance of any specific service configuration of an SRM include the *"average response time"* of a job, *average queue length* of the arrival queue. In addition we examine the *"hit-ratio"* and the *average cost per file retrieval"* of the cache replacement policies. These are formerly defined in section 2. The results reported in this paper are only for the *"average response time"*, the *average cost per file retrieval"* and the

"hit-ratio" for an admission policy we refer to as *"Opt-CacheLoad"* when combined with the least recently used (LRU), cache replacement policy.

1.3 Main Results and Contributions

We focus on a variant of the above problem and then consider a development of an admission policy that we combine with a detailed cache replacement policy for storage resource managers. The main results of this paper are that:

1. An efficient admission policy can be achieved, for a defined dynamically varying value $v(J_i)$ for each job J_i. The admission policy is termed the *OptCacheLoad* policy. We show that *OptCacheLoad* is analogous to the *Budgeted Maximum Coverage Problem* introduced by Khuller et al. [9].

2. Based on the mapping of *OptCacheLoad* admission policy to the *Budgeted Maximum Coverage Problem*, we conclude that the admission policy gives a result that is within a factor of $1 - 1/e$ of the optimal.

3. We present an accurate and detailed framework for evaluating cache replacement policy using finite state machine (FSM), with conditional transitions.

4. Using an FSM model for a combined LRU cache replacement algorithm and *OptCacheLoad* admission policy, we show that the response times of grid jobs serviced at an SRM are considerable improved compared to a combination of a simple first come first serve (FCFS) admission policy with LRU cache replacement policy. These results are obtained for both synthetic and real workloads. The real workload is obtained from the log of file accesses made to a mass storage system.

5. The use of an admission policy also improves the hit-ratio and the average cost per retrieval of cache replacement policies.

The result of this work has applications in a number of areas. First, the result can be integrated as a policy advisory module in the implementations of storage resource managers and storage area networks. Such a policy module, either as an SRM component or directly in coordination with mass storage systems, provides optimization strategy in the use of data grids. The major benefits to be gained are in reduced network traffic, reduced average response times for file requests and optimal resource allocation to meet defined response and deadline objectives under specified local resource management autonomy with little or no global control.

The rest of the paper is organized as follows. We describe the service model of an SRM in section 2. In section 3 we present the theoretical foundation of the *OptCacheLoad* admission policy. Our framework for simulating cache replacement policies is discussed in section 4. Our experimental setup is discussed in section 5 and we present the results of our preliminary studies in section 6. We conclude in section 7 where we also give some directions for future work.

2 Job Service Model of a Storage Resource Manager

Distributed scientific applications often 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* was first used to define a project funded by the European Union that aims at enabling access to geographically distributed computing and storage facilities belonging to different institutions. It has since been used to imply any distributed network infrastructure of storage resources and repositories of huge amounts of data coming from scientific experiments in primarily three different disciplines: High Energy Physics, Biology and Earth Observation Systems. The idea is to support scientific explorations that require intensive computations and analyses of large-scale shared databases across widely distributed scientific communities.

2.1 Servicing of File Requests

The service scenario in a data grid for a typical application is as follows. Grid jobs make requests for a large number of files that reside on a network of distributed tertiary storage systems or mass storage systems. An example of such systems is the IBM's high performance storage system (HPSS) or a storage area network (SAN). The file requests in a job can be for hundreds or thousands of files at the same time. To improve the access performance of the data grid, a middleware component generally referred to as a storage resource manager (SRM) [16], is used to facilitates the sharing of the distributed data and storage resources. An SRM maintains a large capacity disk for caching files of varying sizes that are read from or written to Mass Storage Systems (MSS). Just as mass storage systems form a distributed network of storage resources, SRM's form a network of disk resources for staging files accessed by grid jobs. An SRM generally queues the jobs and subsequently makes decisions as to which job has to be serviced next and which file from the requests of the selected job, must be retrieved into or

transferred from the disk cache. In general, under a workload of shared accesses and high locality of reference, the optimal use of a data grid in data intensive application depends heavily on the policies for data replication, caching, file request forwarding, cache replacement and local servicing (or scheduling) of file requests. We distinguish between data replication and caching. In replication, the information that a file has been staged in a particular disk resource is immediately communicated to some replica management service so that all other storage resource managers can become aware of this. In caching, only the local storage resource is aware of the presence of the file in its disk cache.

2.2 Job Service Policies

We investigate and define a suite of policies that can be selectively combined to enhance or optimize the performance of data accesses in data grids. The various classes of policies of concern are:

Service Policies: The rules that govern the decision for selecting the job whose file request is to be processed next. These are typically defined within the local constraining policies such as, how much disk space a job can use, the number of simultaneous files a job is allowed to process, etc.

Caching Policies: The rules governing the selection of which file, amongst the list of files of the selected job, must be retrieved into the disk cache.

Caching Replacement Policies: The rules that govern the selection of a file to be evicted from the cache when space is needed.

Replication Policy: The rules that govern when, where and which file to replicate in a multi-tier distributed storage environment. This may use some defined constraint on how many replicas and which replicas may be used.

Just as in database management systems, where optimal query processing requires that issues related to, query rewrite, data buffering (or caching), indexing, etc., must be addressed so must corresponding related problems of optimal file accesses in data grids be addressed to achieve a respectable level of performance in data intensive applications. The problems we address are those arising from large scale data accesses in data grids and may be perceived as the relative counterparts of data access problems in distributed database management system.

2.3 Performance Metrics

The performance of admission policies, which may be referred to also as schedules, is measured by the response times of jobs, the average queue length of waiting jobs and sometimes the *"makespan"* of the schedule. 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. Cache replacement policies are typically designed to optimize the *hit ratio* usually by retaining 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). Both policies are predicated on the assumption that a reference stream has a high degree of locality of reference.

Since the goal of caching is to improve the overall performance of jobs serviced by an SRM, we will consider, as our measures for comparisons between different alternative admission policies, the response times, the average cost per file retrieval and the hit ratio of the cache replacement algorithms.

Average response time of a job: The response time of a job is the time between the last service completion time of the file requests of the job and the arrival time of the job. Suppose a workload contains N distinct jobs. Let the job i have a response T_i, then the average response time time denoted by $\bar{T}$ is defined by $\bar{T} = 1/N \sum_i T_i$.

Average Cost Per Reference (ACPR): This metric measures the effectiveness of a caching policy by the average response time per reference. It takes into consideration the total delay in caching files of varying sizes, the varying source delays and the varying transfer times. Suppose for a given workload of a set of $\mathcal{R}$ file references, a subset $\mathcal{R}' \subseteq \mathcal{R}$ of the files are retrieved. Each file $i \in \mathcal{R}'$, that is retrieved, is done at a cost $c_i(t)$, where the cost is measured in time units and is given simply by total time it takes to completely read the file into the disk cache. The average cost per retrieval, denoted by $\bar{c}$ is then defined as $\bar{c} = 1/|\mathcal{R}| \sum_{i \in \mathcal{R}'} c_i(t)$. Consequently an optimal replacement algorithm based on ACPR, implicitly minimizes the response times of file requests. This is a more practical measure for the effectiveness of a cache replacement algorithm for SRMs on the grid.

Hit Ratio: This is given by the ratio of the number of references that encounter cache hits to the total number of file references. This assumes that all files are of the same size and have the same access cost. This assumption is unrealistic in the use of SRMs in data grids. The files have varying sizes and have replicas at different sources with different delays and transfer cost into an

SRM's disk cache. It is easy to envisage a replacement policy that favors only files of small sizes thereby retaining as many files in cache as possible and improving the hit ratio at the expense of high retrieval cost and poor response time whenever large files are referenced. Hit ratio only measures the effectiveness of the use of a cache as the number of hits and does not reflect in any way the effects of source and transfers delays of the files. We include it in the measures considered in this paper simply because it is a popular measure used extensively in the literature on caching.

2.4 Related Works

The concept of data-grids and storage resource management have only recently been given considerable research attention as a result of the need to support large scale scientific experiments - PPDG [8] and GriPhyN [5] - some of which are expected to be operational in three to five years. Storage resource management and some related component prototypes are already in service [16, 14, 12]. These make use of the Globus toolkit [13], that is becoming the de-facto standard software for implementing grid services. These systems still lack the fine tuning required for the optimal operation of real systems. A close analogous environment from which we leverage some experience is in web services. For example, web-caching [2, 17], 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. This is one reason why we introduced a different performance metric for caching in the grid domain as opposed such simple metrics as *hit-ratio* and *byte hit-ratio*. We discuss our machinery for evaluating cache replacement policies in the next section.

3 Theoretical Foundation of Admission Policies

More formally, we are given a set of jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_N\}$, that arrive independently into a queue Q_a, where each job J_i is associated with some value $v(J_i)$, a set of files $\mathcal{F} = \{f_1, f_2, \ldots, f_M\}$ where each file f_j is of size $s(f_j)$ and a cache C of size $s(C)$. The value of a job may measure its priority or the overall importance assigned to the application it represents. Each job J_i, makes requests for a subset $\mathcal{F}_i \subseteq F$, of the files. A job J_i is removed from the queue only after all the files it requested have been serviced. An example of the jobs together with the set of files being requested can be depicted as a bipartite graph as shown in Figure 1.

For a subset $\mathcal{G} \subseteq F$, we denote by $s(\mathcal{G})$ the sum of the sizes of files in $\mathcal{G}$. We will show that our problem involves

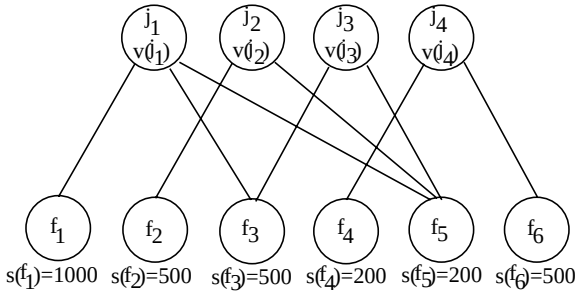

Figure 1. A bipartite graph depiction of a set of jobs and their file requests.

solving the optimization problem, we call *OptCacheLoad* which stands for "*Optimal Cache Loading.*"

OptCacheLoad: Find a subset of files $\mathcal{G} \subseteq F$, which when loaded into the cache C, $(s(\mathcal{G}) \leq s(C))$, maximizes the total value of jobs served by the files in $\mathcal{G}$.

It turns out that *OptCacheLoad* is equivalent to the *Budgeted Maximum Coverage* problem introduced by Khuller et al. [9] that is defined as follows:

Budgeted Maximum Coverage Problem: Given a collection S of sets with associated costs defined over a domain of weighted elements, and a budget L, find a subset $S' \subseteq S$ such that the total cost of sets in S' does not exceed L, and the total weight of elements covered by S' is maximized.

The analogy between the two problems is as follows. The sets in S are our jobs, i.e. $\mathcal{J} \equiv S$, the budget L corresponds to $s(C)$, the available size of our cache C. The weighted elements correspond to our files with weights equivalent to the file sizes $s(f_j)$. The total cost corresponds to the total value of jobs. In [9] it is shown that this problem is NP-hard even for the special case of this problem, where each set has a unit cost. However efficient approximation algorithms were developed in [9], and shown to produce a result bounded from the optimal algorithm by a factor of $1 - 1/e$.

We adopt the following greedy algorithm from [1] for our purposes. Assuming the cache C already contains some files, we can start servicing jobs in Q_a that need these files. The problem is to find an optimal set of files to be loaded next. Intuitively, the benefit of loading a specific file into the cache is proportional to the total value of additional jobs that can be serviced by loading the file but inversely proportional to the size of the file. Let $\mathcal{J}'(f_i)$ be the set of additional jobs that can be serviced by loading file f_i into the cache C. We define for each file f_i its relative value, $v'(f_i)$,

where $v'(f_i) = \sum_{r \in \mathcal{J}'(f_i)} v(r)/s(f_i)$. We note that $v'(f_i)$ changes dynamically with the contents of the cache. The algorithm for admitting a set of files to be loaded into the cache when some cache space of size $s(C)$ becomes available is given below.

Algorithm 1: The OptCacheLoad Algorithm for Admitting a set of files

Data: A set of jobs $\mathcal{J}$; a set of all files requested by $\mathcal{J}$; a cache C with available free space $s(C)$ and a value function $v(J_i)$ expressed for each $J_i \in \mathcal{J}$.

Result: A set of files $\mathcal{G}$, that when loaded into the cache maximizes $\sum_{f_i \in \mathcal{G}}(\sum_{r \in \mathcal{J}'(f_i)} v(r))$.

Initialization;
/* ρ keeps track of size of unused cache space; $\mathcal{J}_s$ keeps track jobs served; $\mathcal{G}$ keeps track of loaded files. */
/* Note that $\mathcal{F} \setminus \mathcal{G}$ is the set of unloaded files */
$\rho \leftarrow s(C)$;
$\mathcal{J}_s \leftarrow \phi$;
$\mathcal{G} \leftarrow \phi$;
while $\rho \neq 0 \wedge \mathcal{F} \setminus \mathcal{G} \neq \phi$ **do**
    Order the files $f_i \in \mathcal{F} \setminus \mathcal{G}$ in non-increasing order of $v'(f_i)$;
    $f_j \leftarrow$ the first file in this ordering that fits in the cache if one can be found ;
    if *found*(f_j) **then**
        Load the file f_j into the cache ;
        $\rho \leftarrow \rho - s(f_j)$; // Update available cache size
        $\mathcal{J}_s \leftarrow \mathcal{J}_s \cup \mathcal{J}'(f_j)$; // update the set of requests that can be serviced
        $\mathcal{G} \leftarrow \mathcal{G} \cup f_j$; // update the set of loaded files
    else
        break;
    end
end

The value $v(J_i)$, of a job can be seen as a quantitative measure of fairness for the jobs and can be defined in a number of different ways. For our purpose and in the experiments conducted in the simulations of this paper, we define $v(J_i)$ simply by the difference in time between the current time and the last recorded time that a file request for a job was made. A large value for $v(J_i)$ indicates that the job has been idle for a long time and should be given a higher priority.

4 The Simulation Framework of Cache Replacement Policies

We note that cache replacement policies have been studied extensively in the literature. These appear in data transfers between computing system's memory hierarchy, database buffer management and in web-caching. Cache replacement models for these situations assume that the request to cache an object is always serviced immediately and once the object is cached, the service on the object is carried out instantaneous. We have not encountered any comparative studies of replacement policies that address long retrieval and processing delays, where the file must be held or pinned in cache for a considerable long time while it is being processed. Almost all models of cache replacement assume instantaneous references to the files, or the cached objects. In the comparative studies of cache replacement policies, in virtual memory, database buffering and web-caching, the object or file references in the workload are serviced strictly in the order of occurrence of the references. Further more the models not only assume that the references made to a cached object is instantaneous but that some file can always be selected for eviction. As a result the literature gives us very simplistic simulation models for the comparative studies of cache replacement policies. Such models are inappropriate in the data grid. We develop and implement an appropriate simulation model that takes into account the inherent delays in locating the file, transferring the file into the cache and holding the file in the cache while it is processed. The sizes of the files we deal with impose these long delays. We capture these in the general setup of our simulation framework.

Figure 2 shows the organization of the information required to simulate the disk cache. There are three data structures for holding the information about the disk cache: a search structure T_1 to hold information about referenced and active files; a data structure T_2 for organizing the information on files that are available for eviction, and a third data structure V_1 to hold the files that are pinned in cache.

A search tree T_1: This is a balanced binary search tree using the file identifier f_i. The nodes of T_1 hold information of all referenced files that are considered to be active. In particular, the nodes hold pointers to the locations of elements that are either in the structure T_2 or in the vector V_1. A status indicator specifies whether the file is considered to be in T_2 or V_1. In addition T_1 holds information of the history of references made since the first reference that caused a node to be created for the file.

A data structure T_2: The elements in the data structure T_2 hold information about files that are in cache but not pinned. The algorithm for cache replacement is evaluated on the non-empty data structure T_2. We choose an efficient implementation of T_2 according to the cache replacement algorithm being considered. For example, T_2 will be implemented as a vector when considering *random* replacement and it will be implemented as a

priority queue data structure when considering *least recently used (LRU), least frequently used (LFU) and greedy dual size (GDS).*

The Vector V_1: The entries in the vector V_1 hold information for files that are the cache and pinned. A file is pinned in cache either because some space have been reserved for it and is in the process of being cached or it is cached and one or more tasks are processing it. Note that due to the sizes of the files, there is a time delay between the initiation and completion of a file caching operation. In particular, an element in V_1 maintains a count of the number of pins placed on the corresponding file of that element. Each time a task completes, it decrements the pin count by one and whenever the pin count becomes zero, the entry is removed and the appropriate node, corresponding to the file, inserted into T_2.

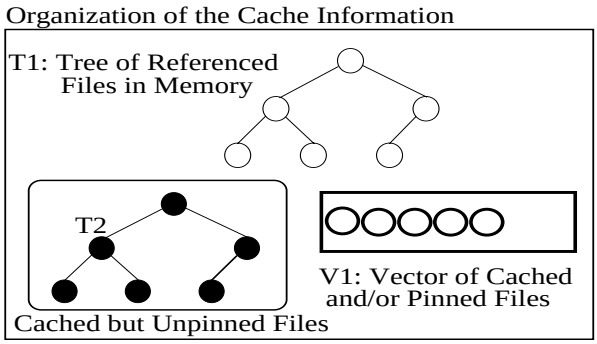

Figure 2. The Organization of Information to Simulate File Caching in an SRM

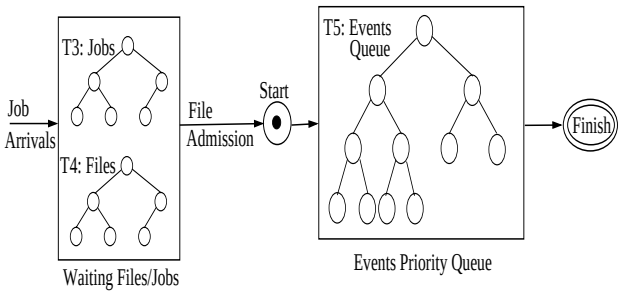

Figure 3. A Simulation Model of an SRM at a Local Host

A file that is referenced, cached, processed and eventually evicted from the cache is considered to undergo some state changes. The information about the state of a file is maintained in the nodes of T_1. A file that is referenced can be in one of five possible states. The various states of a file are: *Not-In-Memory or Not-Referenced (S_0), In-Memory-But-Not-In-Cache (S_1), In-Cache-But-Not-Pinned (S_2), Space-Reserved-But-Not-Cached (S_3)* and *In-Cache-And-Pinned (S_4).* The table 1 shows an *Exit State (S_5)* which is equivalent to S_0. Since we need to retain some history of the references made to a file even when it is not in cache the memory resident search structure T_1, continues to maintain the relevant node of an evicted file. However, the memory resident information in T_1 does not grow indefinitely. At periodic times, say every 5 days (this is determined by the job arrival rate of the workload), the nodes corresponding to all files that have not been referenced in the last five days are cleared. A special event, *Clear-Aged-file (E_5),* when it occurs, causes the nodes of all such files to be deleted. Any future reference to a file, whose node has been cleared, would reinsert a new node entry and initiate a new accumulation of historical information.

The events affecting the state changes of the files are caused by the actions of the tasks that are invoked by the jobs. Figure 3 illustrates some of the details of the simulation framework used in processing jobs at an SRM host. Jobs that arrive at a host are maintained in the search structure T_3. The files requested by the jobs are maintained in a search structure T_4. Besides other relevant information, each node of T_3 holds a list of identifiers of the files being requested by the job. Similarly each node of T_4 corresponds to a unique file and maintains a list of the identifiers of all the jobs requesting that file.

A file admission policy is used to select a file to be retrieved next. If the file selected to be admitted has no information in T_1, an appropriate node is created and inserted into T_1. Upon making a decision on the file to be brought into the disk cache, each of the jobs associated with the file initiates a task token that is inserted into the event queue T_5. Each task token is uniquely identified by the pair of values of the job and file identifiers and subjected to five distinct events at different times. These events are: *Start-Caching (E_0), End-Caching (E_1), Start-Processing (E_2)* and *End-Processing (E_3).* The entire activities within this framework are executed as a discrete event simulation. The activities of the simulation may be summarized by the finite state machine, with conditional transitions, shown in Table 1. The equivalent state transition diagram is given in figure 4.

Each entry in the table defines a function $\psi_{\{i,j\}}$ that is evaluated and a possible state transition is made depending on the outcome of the evaluation. For example, if a *Start-Caching* event occurs on a file that is in state S_1, the function $\psi_{\{1,1\}}$ is computed and a conditional transition is made into state S_3. $\psi_{\{1,1\}}$ involves reserving space in the cache and then initiating a transfer operation to bring the file into the disk cache. The evaluation of $\psi_{\{1,1\}}$ could trig-

State		Event Types on Files						
		E_0 Admit File	E_1 Start Caching	E_2 End Caching	E_3 Start Processing	E_4 End processing	E_5 Cache Eviction	E_6 Clear Aged Files
S_0	Start: File Not in Memory	$\psi_{\{0,0\}}() / S_1$						
S_1	File in Memory but not Cached		$\psi_{\{1,1\}}() / Cond(S_3)$					$\psi_{\{1,6\}}() / Cond(S_5)$
S_2	File in Cache but not Pinned		$\psi_{\{2,1\}}() / S_4$				$\psi_{\{2,5\}}() / Cond(S_1)$	
S_3	Space Reserved but not Cached		$\psi_{\{3,1\}}() / S_3$	$\psi_{\{3,2\}}() / Cond(S_4)$				
S_4	File in Cache and Pinned		$\psi_{\{4,1\}}() / S_4$	$\psi_{\{4,2\}}() / S_4$	$\psi_{\{4,3\}}() / S_4$	$\psi_{\{4,4\}}() / Cond(\{S_2 \mid S_4\})$		
S_5	Exit: File Removed from Memory							

Table 1. The Final State Machine with Conditional Transition of File Requested

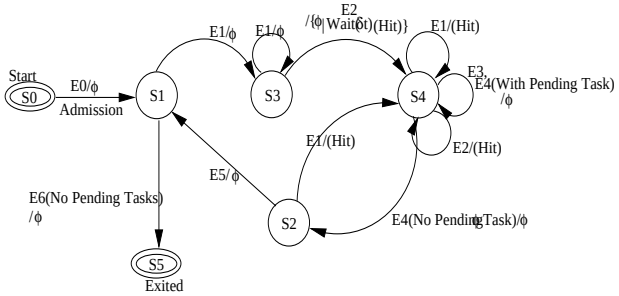

Figure 4. A Finite State Machine Diagram with Conditional Transitions

ger an event of type E_5 to evict some files from the cache until enough free space becomes available. After space is reserved the file moves into state S_3. The task token is reinserted into the event queue T_5 after setting its next event type to *End-Caching* and its next event time to the time that the file caching completes. If no space can be reserved, the file request may be discarded. However, if we ensure that the admission policy guarantees that there would be free space available before the file is admitted then, no file request can be discarded.

The FSM above represents only the model of one site. Such FSMs may be configured into a multi-tier network to represent a multi-tier network of SRM sites. We expand to single site framework to multi-tier networks where file replication decisions may also be made and request can be forwarded to other sites in our future work.

5 Experimental Setup

We conducted some experiments to determine the impact of the *OptCacheLoad* admission policy on the response times of jobs submitted at an SRM host. The assumption here is that the tasks executed by the jobs are I/O-bound jobs as opposed to CPU-bound (or compute intensive) jobs. We conducted performance tests using the simulation framework to evaluate cache replacement policies. Our implementation is a straight forward translation of the FSM, with conditional transitions, to a C++ code. Since our focus here is on the impact of the admission policy, we consider this in conjunction only with the LRU cache replacement policy. Results of extensive comparisons of different cache replacement policies only using the same simulation framework for realistic workloads are discussed in [11]. When a file is cached, the tasks of a job process the file at a rate of 10 MBytes per second.

5.1 Workload Characteristics

We subjected the simulation model to two types of workloads in our experiments: a real workload and a synthetic workload. The real workload is the log of about 6 months of

file accesses to the Jasmine mass storage system at the Jefferson National Accelerator Facility (JNAF). The file sizes ranged from about 500 KBytes to about 6 GBytes per file. The jobs of file requests contain batched requests for 1 to about 1000 files per job. There is however very low locality of file references. By locality of reference we mean the occurrence of references for the same file in jobs that are close to each other in the workload. For the synthetic workload, we extracted statistics of the real workload, such as the average file size, the job inter-arrival times, etc., and generated a workload based on Poisson job arrivals but with a high locality of reference. For both the JNAF workload and the synthetic workload, the file accesses were read-only.

5.2 Simulation Runs

The simulation runs were carried out on a Redhat Linux machine with 512 MBytes of memory. We evaluated the performance metrics of the average response time per job, the average cost per retrieval of a file and the hit ratio for LRU under two alternative configurations. The first configuration executes the workloads by applying the *OptCacheLoad* admission policy to select the set of files to be admitted. The second configuration executes the workloads according to a FCFS admission policy. For each configuration and for each workload a number of runs were done with cache sizes varying from 200 Gigabytes to about 3 Terabytes. For each run and for each cache size, we applied a variance reduction method by averaging the statistics that we compute independently for 5 to 7 segments of the workload. The results of the experimental runs are discussed next.

6 Preliminary Results

Figures 5a and 5b show the graphs of the response times for the synthetic and JNAF workloads respectively that were obtained from servicing jobs using the simulation framework of an SRM. For the synthetic workload, 500 GBytes of disk cache size corresponds to about 0.5% of the data processed and 800 GBytes of cache size corresponds to about 1.0%. In the JNAF workload, 1 Terabyte of disk cache size corresponds to about 0.65% of the data processed and 2 Terabytes of a cache size corresponds to about 1.3% of the data processed. Clearly for most practical disk cache sizes $s(C) \leq 1$ Terabyte, there is a very significant gain in response time when *OptCacheLoad* admission policy is used.

In [11], we defined the average cost per retrieval as the appropriate measure for the performance evaluation of disk cache replacement policies in a data grid. It is noteworthy, from the graphs of figures 5c and 5d, that not only does the use of the *OptCacheLoad* admission policy improve the response time of jobs but the average cost per retrieval is also improved for any practical limit of cache sizes. From figures 5e and 5f, we observe similar improvements for the hit-ratios when *OptCacheLoad* admission policy is used. In fact the same relatively results are obtained for other cache replacement policies. Due to space limitation we do not report these.

7 Conclusion and Future Work

Unlike page caching in virtual memory management or page buffering in data base management systems, file caching onto disks is not relatively instantaneous. We have developed and implemented a simulation framework for investigating both file scheduling and cache replacement policies that may be used to govern I/O-bound job execution on data grids. Using this simulation model of an SRM host, we have demonstrated that a good admission policy, that takes into account the file requests of the waiting jobs, significantly improves the average response time of a job, the average cost per retrieval and the hit ratio of the cache.

The results reported in this paper are only preliminary results of an ongoing work to implement realistic simulation model of SRMs and subsequently a network of SRMs that are deployed in data grids. Future work extensions of our current research include:

- The characterization of workloads of file accesses to different mass storage systems. These concern the logs of data accesses to mass storage systems used in different scientific collaborations.
- An implementation of a Policy Advisory Module (PAM), including the ability to configure different policy options, that can be integrated with deployed SRMs.
- The use of a storage resource manager emulator to extensively study the performance of data accesses of real workloads, under various combinations of admission, caching, cache replacement, replication, and replica replacement policies.
- An emulator for multi-tier adaptive file caching and replication in data grids.

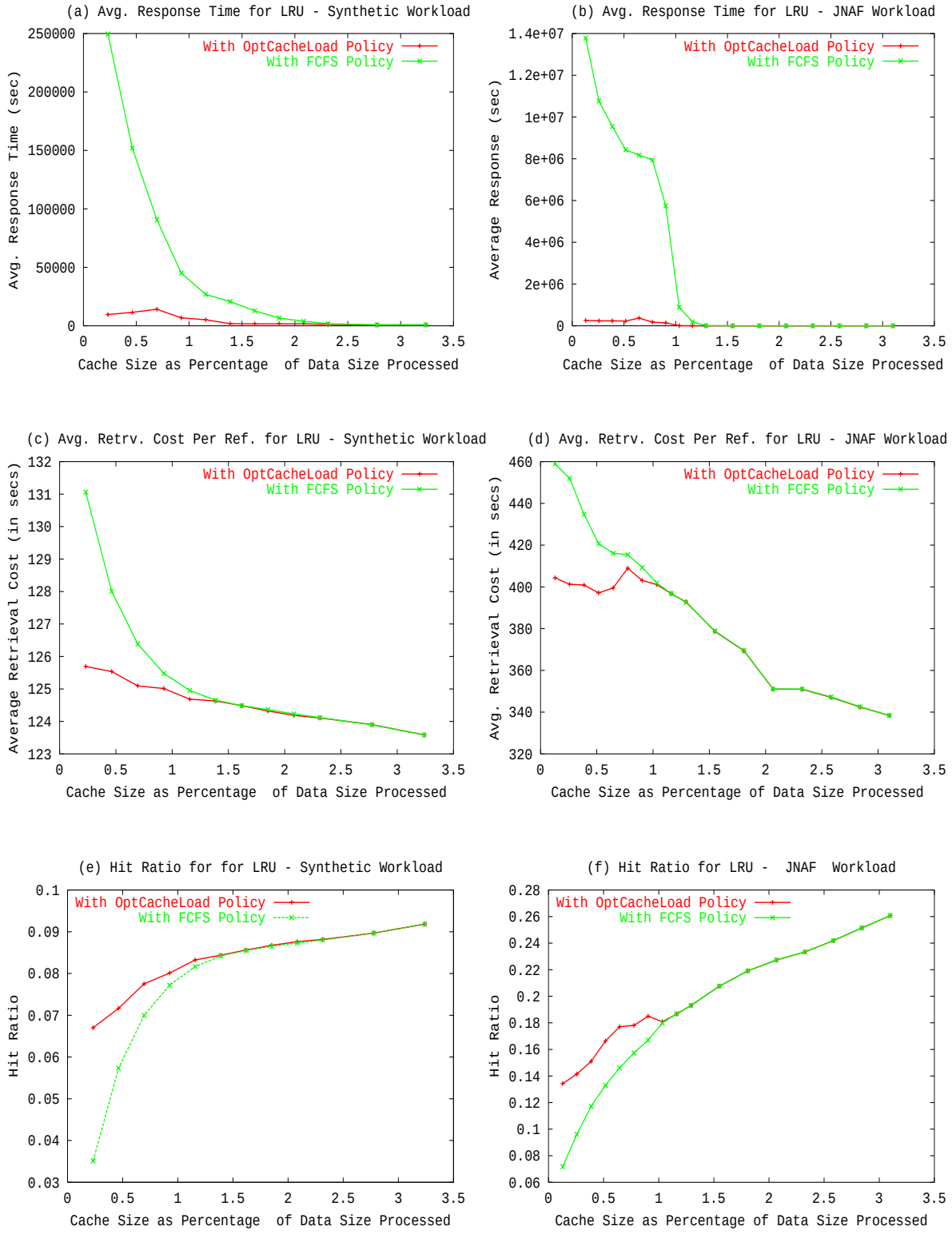

Figure 5. Effect of *OptCacheLoad* on Jobs Serviced at an SRM using LRU Cache Replacement Algorithm

References

[1] K. Aida, A. Tekefusa, H. Nakada, S. Matsuoka, S. Sekiguchi, and U. Nagashima. Performance evaluation model for scheduling in global computing systems. *Int'l. J. of High Perform. Comput. Appl.*, 14(3):268–279, 2000.

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

[3] H. Casanova. SIMGRID: A toolkit for the simulation of application scheduling. In *Proc. of the First IEEE/ACM Int'l. Symp. on Cluster Comput. and the Grid (CCGrid 2001)*, 2001.

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

[5] GriPhyN: The Grid Physics Network Collaboration. http://www.phys.ufl.edu/ avery/griphyn/.

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

[7] ESG: The earth science grid. http://www.scd.ucar.edu/css/esg/.

[8] PPDG: The particle physics data grid. http://www.ppdg.net/.

[9] S. Khuller, A. Moss, and J. S. Naor. The budgeted maximum coverage problem. *Information Processing Letters*, 70(1):39–45, 1999.

[10] H. Lamehamedi, Z. Shentu, B. Szymanski, and E. Deelman. Simulation of dynamic data replication strategies in data grids, June 2002.

[11] 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.

[12] GDMP: grid data mirroring package. http://project-gdmp.web.cern.ch/project-gdmp/.

[13] The globus project. http://www.globus.org/.

[14] 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.

[15] K. Ranganathan and I. T. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proc. of 11th IEEE Int'l. Symp. on High Perform. Distrib. Comput. (HPDC-11)*, Edinburgh, Scotland, July 2002.

[16] 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.

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