Filecules: A New Granularity for Resource Management in Grids

by

Shyamala Doraimani

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Adriana Iamnitchi, Ph.D.
Gabriele Garzoglio, Ph.D.
Ken Christensen, Ph.D.

Date of Approval:
March 26, 2007

Keywords: Caching, Data Management, File Grouping, Grid, Scientific Computing,
Workload Characterization

© Copyright 2007, Shyamala Doraimani

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FILECULES: A NEW GRANULARITY FOR RESOURCE MANAGEMENT IN GRIDS

Shyamala Doraimani

ABSTRACT

Grids provide an infrastructure for seamless, secure access to a globally distributed set of shared computing resources. Grid computing has reached the stage where deployments are run in production mode. In the most active Grid community, the scientific community, jobs are data and compute intensive. Scientific Grid deployments offer the opportunity for revisiting and perhaps updating traditional beliefs related to workload models and hence reevaluate traditional resource management techniques.

In this thesis, we study usage patterns from a large-scale scientific Grid collaboration in high-energy physics. We focus mainly on data usage, since data is the major resource for this class of applications. We perform a detailed workload characterization which led us to propose a new data abstraction, *filecule*, that groups correlated files. We characterize filecules and show that they are an appropriate data granularity for resource management.

In scientific applications, job scheduling and data staging are tightly coupled. The only algorithm previously proposed for this class of applications, Greedy Request Value (GRV), uses a function that assigns a relative value to a job. We wrote a cache simulator that uses the same technique of combining cache replacement with job reordering to evaluate and compare quantitatively a set of alternative solutions. These solutions are combinations of Least Recently Used (LRU) and GRV from the cache replacement space with First-Come First-Served (FCFS) and the GRV-specific job reordering from the scheduling space. Using real workload from the DZero Experiment at Fermi National Accelerator Laboratory, we measure and compare performance based on byte hit rate, cache change, job waiting time, job waiting queue length, and scheduling overhead.

Based on our experimental investigations, we propose a new technique that combines LRU for cache replacement and job scheduling based on the relative request value. This technique incurs less data transfer costs than the GRV algorithm and shorter job processing delays than FCFS. We also propose using filecules for data management to further improve the results obtained from the above LRU and GRV combination.

We show that filecules can be identified in practical situations and demonstrate how the accuracy of filecule identification influences caching performance.

CHAPTER 1

INTRODUCTION

Sustained effort is ongoing to support various scientific communities and their large-scale data-sharing and data-analysis needs through a distributed, transparent infrastructure. This effort is part of a research area known as Grid computing, an area whose primary objective is to provide an infrastructure for seamless and secure access to a globally distributed set of shared software and hardware resources. This infrastructure necessarily includes components for file location and management as well as for computation and data transfer scheduling.

There is little information available on the specific usage patterns that emerge in these data-intensive, scientific communities. This research analyzes usage patterns in a typical Grid community followed by an experimental investigation of how these patterns can be exploited for data management. We analyze the characteristics of a production mode data-intensive high-energy physics collaboration, the DZero Experiment [2], hosted at Fermi National Accelerator Laboratory (FermiLab).

Figure 1.1 shows 3 sites that participate in the DZero Experiment. Each site has storage and computing resources. The storage resources are managed by Storage Resource Managers (SRM), which take care of transferring data from storage at other sites to execute a job. SRM has a disk cache where the data is stored. In this thesis, we discuss techniques that reduce the amount of data transferred between sites and also increase the throughput at each site.

In the DZero Experiment, users submit requests to analyze data to obtain physics measurements. Hundreds of jobs are submitted each day by scientists from all over the world. These jobs perform read-only operation on the data. Our analysis shows that the usage of data in the DZero Experiment exhibit good temporal locality and that scientific

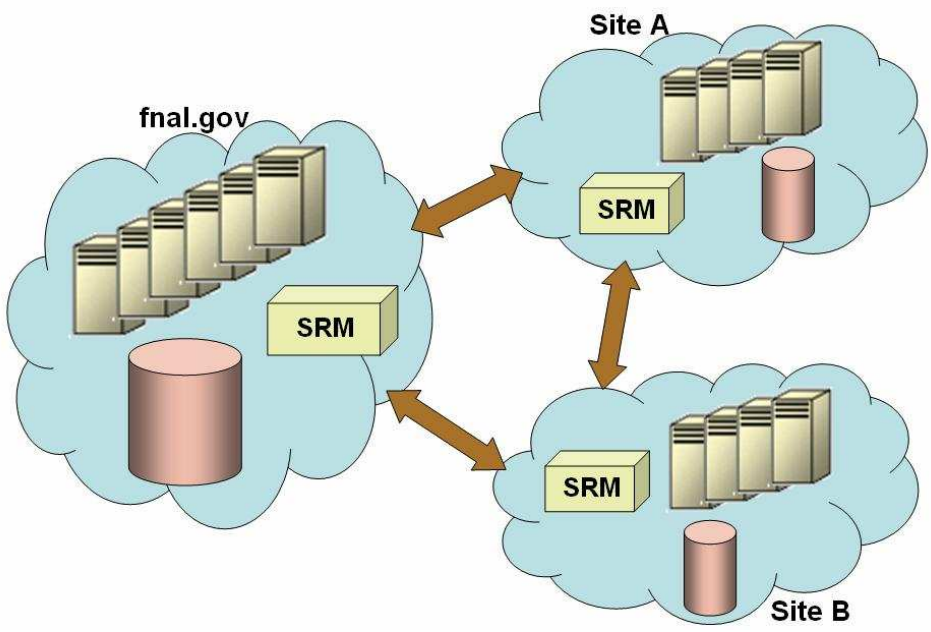

Figure 1.1 The System Configuration

data usage translates into requests for groups of correlated files. This fact suggests a new granularity for data management, *filecules* [fil'-eh-kyul'], which can be exploited to design solutions that significantly outperform the traditional solutions based on single file granularity.

We define a filecule as an aggregate of one or more files in a definite arrangement held together by special forces related to their usage. We study the properties most relevant to data management using filecules, namely size, popularity and lifetime. We compare these characteristics with the corresponding characteristics of data from traditional systems (such as file systems and the Internet). We also study the correlation between these properties in order to see if size or lifetime of filecules can be used to make decisions about data management. For example, if there is strong correlation between size and popularity, cache replacement algorithms that evict files based on file sizes can be used in these systems.

The nature of jobs in Grid are such that all the data requested by a jobs needs to be located on disk before the job can run. Thus job scheduling and loading data on disk are tightly coupled. We explore the space of algorithmic design for caching algorithms along two intertwined dimensions: cache replacement and reordering the jobs in the waiting

queue. We compare techniques that are a combination of one cache replacement and one job reordering algorithm. These techniques are combinations of Least Recently Used (LRU) and Greedy Request Value (GRV) [42, 44, 43] from the cache replacement space with First-Come First-Served (FCFS) and the Greedy Request Value-specific job reordering from the scheduling space. We use GRV for our comparison because it is the only cache replacement policy that has been proposed for jobs in Grids that use multiple files. The GRV algorithm uses a job scheduling algorithm that changes the order in which jobs are processed based on the contents of the cache.

The algorithms are compared based on the following metrics: byte hit rate, percentage of cache change per job, job waiting time, queue length and number of computations to schedule a job. Byte hit rate per job is a measure of the effectiveness of the cache replacement policy. Percentage of cache change per job shows how much data is transferred in and out of the cache in order to process a job. This is particularly relevant in a grid-like scenario where data transfer is time consuming due to wide-area transfers and size of data (multi-GB or more). Delay in data transfer causes delay in job execution. If the data prefetched is utilized effectively, the percentage of cache change per job will be low. Job waiting time and queue length shows the effectiveness of the job scheduling algorithm. The number of computations to schedule a job is a measure of scheduling overhead.

Our results show that Filecule LRU provides better byte hit rates than GRV. Also based on our results, we identify a new combination of LRU cache replacement with GRV job scheduling algorithm that takes advantage of temporal locality in the workload and the contents of the cache. The GRV job scheduling policy improves the throughput of the system by scheduling jobs that utilize the contents of cache. When the combination of LRU cache replacement algorithm with job scheduling using GRV is used with filecules for data prefetching, it may provide significant improvements to hit rates while reducing the amount of data that needs to be replaced in cache.

In order to understand the impact of history of jobs in identifying filecules that improve the performance of the system, we study the variations in byte hit rates and volume of cache change for different lengths of history. We see that with increasing history lengths, the filecules are closer to those identified with the entire workload. But the time taken

to identify these filecules is long. With small history lengths, the filecules identified are large in size. Finally we discuss the impact of this new granularity for other resource management services.

To summarize, this thesis:

1. Provides a quantitative evaluation of user behavior in terms of data usage patterns and compares it with traditional models from the literature (Chapter 4). Our analysis shows that while some traditional models work for the DZero workload (such as file popularity observed in web proxy [11] and web client [15]), others are inappropriate (such as file sizes observed in Windows file system [25], P2P [52], Internet [9, 24]). This study provides important information about user requests in the past which can be utilized to predict future data requests and hence reduce delays in processing their requests.

2. Characterizes the properties of filecules (Chapter 4).

3. Discusses the impact of the pattern we discovered from our trace analysis on resource management (Chapter 4).

4. Proposes and evaluates experimentally a new combination of cache replacement policy and job scheduling (Chapter 5).

5. Analyses the impact of size of window history used for filecule identification (Chapter 6). These results show that relatively short history is sufficient for identifying file groups that benefit cache performance. We also observe that a sliding window adapts better to changing usage patterns.

CHAPTER 2

WORKLOAD DESCRIPTION

In Grid terminology [26], the DZero Experiment [2] is a virtual organization consisting of hundreds of physicists in 70+ institutions from 18 countries. It provides a worldwide system of shareable computing and storage resources that can be utilized to solve the common problem of extracting physics results from several Petabytes of measured and simulated data. The workload analyzed in this thesis is from this production-mode data-intensive high-energy physics collaboration, the DZero Experiment. In this chapter, we provide details about the workload and the intuition that led us to propose a new granularity for data management, filecules.

2.1 Data Management in Grids

Storage Resource Managers (SRM) [51] are middleware components that provide space allocation and data management on the Grid. The Grid uses heterogeneous storage resources. SRMs reserve and schedule storage resources by providing standardized uniform access to these heterogeneous storage resources. SRMs have disk caches which can range from few hundreds of gigabytes to tens of terabytes. For example, disk caches vary from 1 TB to 5 TB in DZero, are up to 150 TB [27] in CDF [1] and about 70 TB [27] in DESY [3]. The disk caches store data that are requested by clients and thus mask failures due to link failures. Among many other responsibilities, SRM administer two policies: job scheduling and data caching.

2.2 Data in the DZero Experiment

Modern high-energy physics experiments, such as DZero, typically acquire more than 1 TB of data per day and move up to ten times as much. To give an example, during the

past year the more than half a petabyte of data was stored at Fermi National Accelerator Laboratory. Aside from the stream of data from the detector, various other computing activities contribute to the 1 TB of derived and simulated data stored per day. In this system, data files are read-only and the typical jobs analyze and produce new, processed data files.

Three main activities take place within the DZero Experiment: data filtering (called data reconstruction in the DZero terminology), the production of simulated events, and data analysis. This third activity mainly consists of the selection and the statistical study of particles with certain characteristics, with the goal of achieving physics measurements. The first two activities are indispensable for the third one. During data reconstruction, the binary format of every event from the detector is transformed into a format that more easily maps to abstract physics concepts, such as particle tracks, charge, spin, and others. The original format is instead very closely dependent on the hardware layout of the detector, in order to guarantee the performance of the data acquisition system, and is not suitable for data analysis. On the other hand, the production of simulated events, also called Monte Carlo production, is necessary for understanding and isolating the detector characteristics related to hardware, such as the particle detection efficiency, or to physics phenomena, such as signal to background discrimination. Tracing system utilization is possible via a software layer (SAM [38], [55]) that provides centralized file-based data management. The SAM system offers four main services: first, it provides reliable data storage, either directly from the detector or from data processing facilities around the world. Second, it enables data distribution to and from all of the collaborating institutions. Third, it thoroughly catalogs data for content, provenance, status, location, processing history, user-defined datasets, and so on. And finally, it manages the distributed resources to optimize their usage and enforce the policies of the experiment.

SAM categorizes typical high energy physics computation activities in application families (reconstruction, analysis, etc.). Applications belonging to a family are identified by a name and a version. This categorization is convenient for bookkeeping as well as for resource optimization. Due to the data intensive nature of the high energy physics domain, applications almost always process data. Such data is organized in tiers, defined according

Table 2.1 Data in Workload from Jan 2003 to Mar 2005

Number of jobs	234,069
Number of users	561
Number of data tiers	32
Number of files	1,134,086

to the format of the physics events. Relevant data tiers, some of which are discussed in this work, are the raw, reconstructed, thumbnail, and root-tuple tiers. The raw tier identifies data coming directly from the detector. The reconstructed and thumbnail tiers identify the output of the reconstruction applications, in two different formats. The root-tuple tier identifies typically highly processed events in root format [19] and are generally input to analysis applications. For the data handling middleware, an application running on a dataset defines a job. Jobs are initiated by a user on behalf of a physics group and typically trigger data movement.

2.3 The DZero Workload

The studies presented in this research utilize data from the SAM data processing history database between January 2003 and March 2005. Two types of traces have been selected for our studies: file traces and application traces. File traces show what files have been requested with every job run during the period under study. These traces are used to study the presence of filecules in the DZero computing activity. Application traces list summary information for the jobs. The information includes metadata for the application (application name, version, and family), for the dataset processed (data tier). The application traces also contain general data, such as the user name and group that initiated the job and the location (node name) and start/stop time of the job. Table 2.1 shows the quantitative details of the workload data.

Among the 234,069 jobs submitted during the period of January 2003 and March 2005, 113,454 jobs used data from reconstructed, root-tuple and thumbnail data tiers. Out of these jobs, only 113,062 jobs have information about files requested by the job. These jobs were used for disk cache simulations. Table 2.2 shows details about the workload used for the simulations. The total number of file requests generated by these jobs is 11,568,086.

Table 2.2 Characteristics of Traces Analyzed Per Data Tier

Data tier	Users	Jobs	Files	# of file requests	Input/Job (GB)		Time/Job (hours)	
					Avg.	Std.dev.	Avg.	Std.dev.
Reconstructed	304	17,552	507,796	1,770,176	34	285	11.08	38.52
Root-tuple	51	1,226	59,923	468,176	85	115	14.19	28.93
Thumbnail	440	94,284	428,508	9,329,734	50	319	8.08	28.83

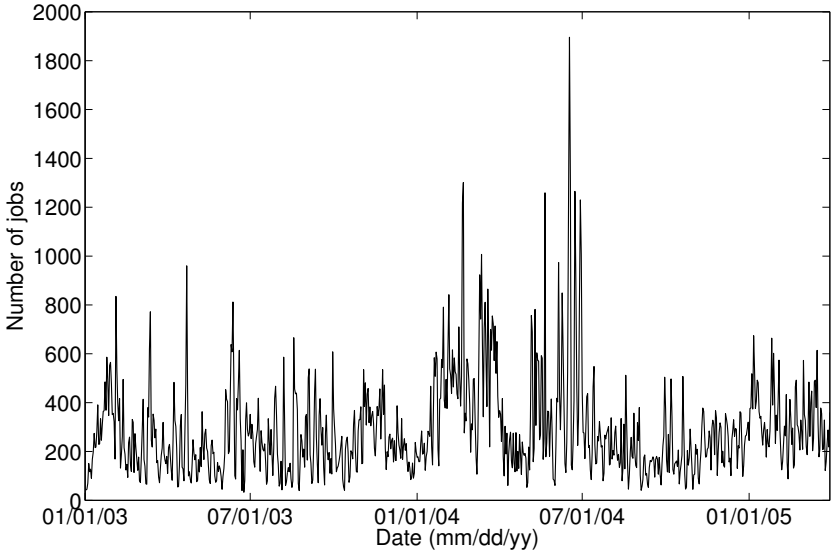

Figure 2.1 Number of Jobs Per Day

2.4 Number of Jobs Per Day

Figure 2.1 shows the number of jobs that were submitted each day. The figure and Table 2.3 shows that hundreds of jobs are submitted on any given day.

Figures 2.2 and 2.3 show that the jobs use multiple files. The distribution shown in Figure 2.3 is heavy tailed. But it is also important to note that the average number of files per job is 102 and the median value is 12. In order for jobs to run, these group of files need to be present all at the same time on the local disk.

Table 2.3 Data Requested Per Day

Category	Mean	Median
Number of jobs per day	137	107
Number of files per job	102	12
Number of file requests per day	15,199	10,853
Number of distinct files requested per day	9,534	7,318
Data accessed per day (TB)	6.86	5.08
Distinct accessed per day (TB)	4.54	3.45

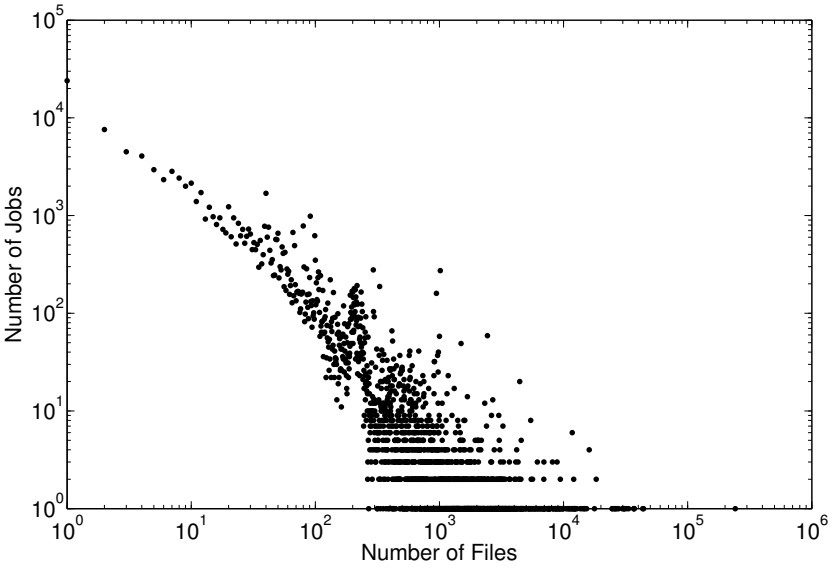

Figure 2.2 Number of Files Per Job

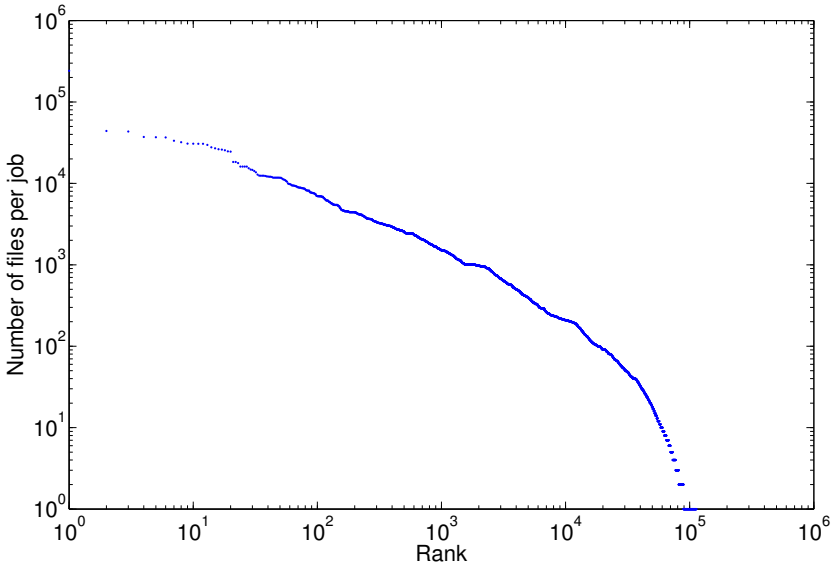

Figure 2.3 Distribution of the Number of Files Per Job

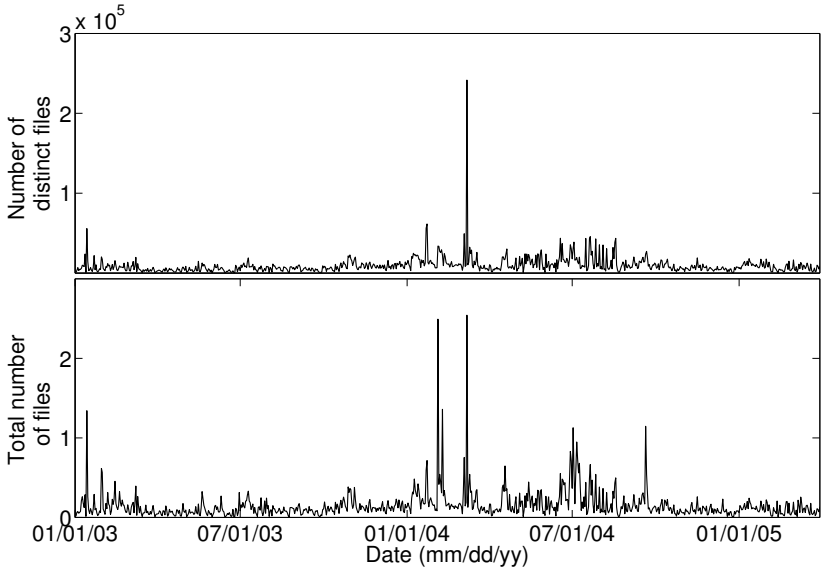

Figure 2.4 Number of Total Files and Distinct Files Requested Per Day

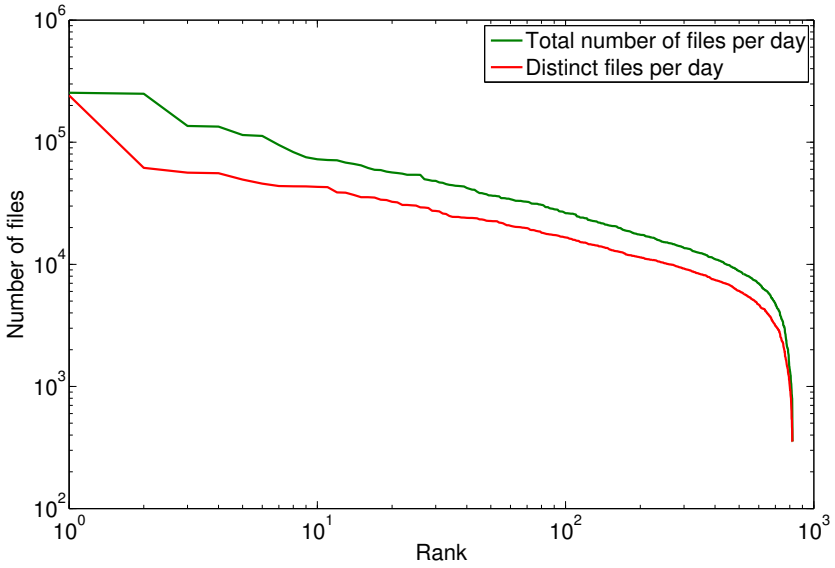

Figure 2.5 Distribution of Total Number of Files Per Day and Distinct Files Per Day

2.5 Data Accessed Per Day

Figures 2.4 and 2.5 show that the distinct number of files accessed per day is smaller than the total number of files accessed on a day. This indicates that some of the files are requested by more than one job.

Figures 2.6 and 2.7 show similar results with the number of distinct bytes accessed being smaller than the total number of bytes. Table 2.3 show that on an average, 37.3% of files requested on a day are accessed twice which equals to about 33.8% of repeated bytes.

Another important factor is that jobs can run for more than a single day. The plots here show the usage based on the start date of the job. If the plot is extended to include two days and so on, the overlap of the number of bytes repeated during that period will be more pronounced.

2.6 Summary

The data presented in this chapter shows that jobs requests multiple files and this leads to tens of terabytes of data accessed per day. There is a significant percentage of data that is repeated each day (Refer Table 2.3). This led us to the intuition that there

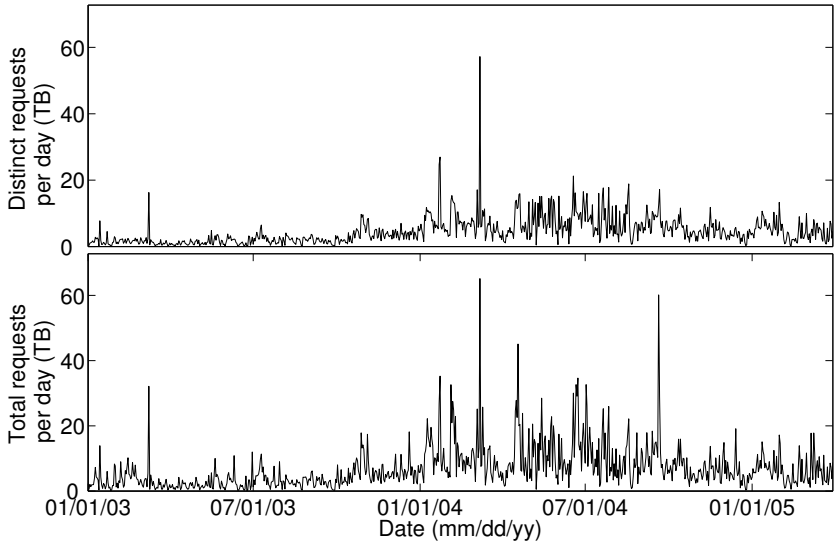

Figure 2.6 Total Number of Bytes Per Day and Distinct Bytes Per Day

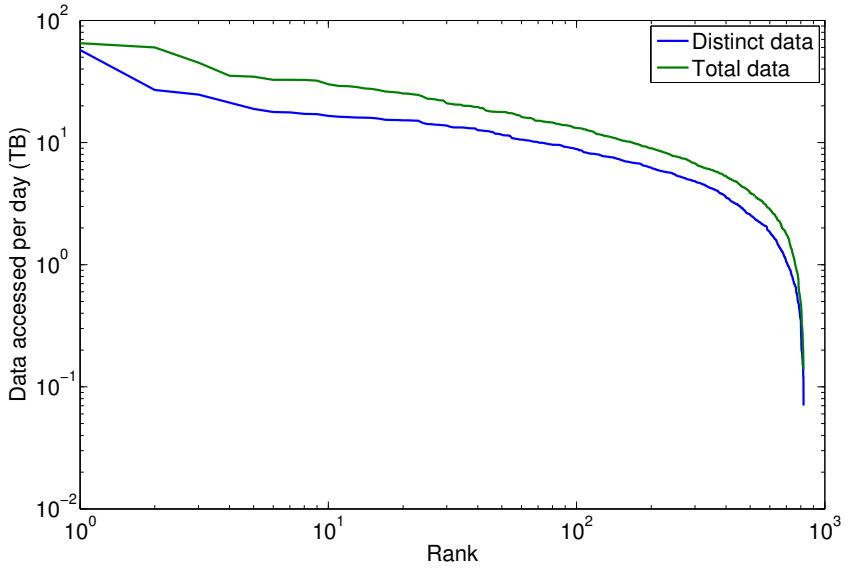

Figure 2.7 Distribution of Total Number of Bytes Per Day and Distinct Bytes Per Day

are relationships between files. In Section 5.1, we show that the usage patterns exhibit good temporal locality. This existence of temporal locality in usage patterns combined with grouping data based on the relationships between files led us to experiment methods that exploit these behaviors to reduce volume of data transferred between nodes in grid and reduce delay to schedule job for execution.

CHAPTER 3

RELATED WORK

The nature of jobs submitted in a system influences system design. Before building a system, it is hard to characterize the nature of jobs. Hence, assumptions are made about the job characteristics by deriving similarities with other systems. Once the system is deployed and running, it provides the opportunity to test the assumptions made earlier and make necessary changes to improve system performance. Solutions that took advantage of observed patterns include file location mechanisms that exploit the relation between stored files [23], information dissemination techniques [34] that exploit overlapping user interests in data [35], and search algorithms [7] adapted to particular overlay topologies [49]. Study of usage patterns also enables creating models for the systems that can be used for analysis without disturbing the actual system.

Section 3.1 details previous work on analyzing file request data from web servers, proxy servers and browsers, Windows and Unix file systems and P2P file sharing systems. Comparing the characteristics of DZero workload with those of traditional file systems can identify models in traditional systems which can be used to analyze the DZero workload. We compare the size, popularity and lifetime characteristics of data in DZero with those of web requests, Windows and Unix file systems, and the more recent file-sharing systems.

Identification of filecules involve grouping of files based on relationships between files. Section 3.2 discusses other methods used to identify relationships between files and how our grouping methodology is different from prior work.

Section 3.3 discusses prior work that describes stack depth analysis and how it is used to test the temporal locality of a workload. We use the same methodolody to test temporal locality in DZero workload. Based on results from stack depth analysis, we choose the LRU

cache replacement algorithm for our simulation. Section 3.4 and 3.5 describes prior work on various cache replacement algorithms.

3.1 Workload Characterization

This section provides details about prior work on workload characterization. Previous work that characterize workloads from the Internet, file systems, P2P file sharing systems and data grid are described.

Web Requests

Web requests have been studied at various end points: web servers, proxy servers and browsers. Barford et al. [15] studied the web-client traces from 2 workstations in Boston University's Computer Science department. File sizes were found to follow lognormal distribution with Pareto tails. A majority of the requests were targeted at a small set of files.

Web server workloads have been studied extensively in [9], [6] and [14]. In [9], Almeida et al. analyzed the logs from the NCSA Web Server at the National Center for Supercomputing Applications, the SDSC Web Server at San Diego Supercomputer center, the EPA Web server at Research Triangle Park, NC and the web server at the Computer Science department at Boston University. The file popularity was found to follow Zipf-like distribution. In [6], Archarya et al. studied the web server logs from Luleå University of Technology, Sweden. The file sizes were found to be concentrated closer to the mean file size. In [14], 6 different web server data sets were analyzed to identify characteristics common to all the workloads. 3 academic, 2 scientific research organizations, 1 ISP were used in the study. There was a small number of very small and very large files. Only 10% of the files were larger than 100 KB. The distribution of file size had a Pareto tail.

In [24], Cunha et al. studied the traces from browser logs of the Boston University CS department. The code of Mosaic browser was modified to log requests from users. The logs amounted to a period of 2 months. File sizes were observed to follow Pareto distribution.

More files are small in size. File popularity followed Zipf distribution. The relation between file size and file popularity was found to be inversely correlated.

In [11], Arlitt et al. studied the traces from a web proxy within an ISP. The traces amounted to a period of 5 months of activity. File-size distribution was found to be heavy-tailed. 90% of the files contributed only to 51% of the total size. 40% of the total size was due to a few large files. File popularity was also heavy-tailed. 37% of files received 78% of the requests while 63% of the files were requested only once. Small files were requested more often than the large files. The popularity follows a Zipf-like distribution. File lifetimes are long for a few files. An active set of one day's file requests was observed over a period of 5 months. About half the files became unpopular on the next day. Further changes to the active set were more gradual (10% reduction each month). About 20% of the day's files were used actively even after 5 months.

In [18], Breslau et al. analyze web proxy cache traces from different sources. They show that the page request distribution follows a Zipf-like distribution. They also show that there is weak correlation between page size and popularity.

File Systems

In [25], Douceur and Bolosky have studied the characteristics of Windows file systems at the Microsoft Corporation. The mean file sizes in these file systems varied from 64 KB to 128 KB. But the median file sizes was just 4 KB. This indicates that there were a lot of small files. The file size distribution follows log-normal. The high mean is influenced by the existence of a few large files. The file lifetimes were observed to follow a hyper-exponential distribution. Similar observations were made by Vogels in [56].

In [37], Gordoni analyzes the file sizes from different unix systems. He identified that the systems either have a lot of small files or have a few big files. Hence he suggests using different strategies when handling small and big files.

In [54], Tanenbaum et al. study the file size distribution on Unix system. They analyzed the file sizes on the Unix machines at the Computer Science department of the Vrije Universiteit during the 2005 and showed that the median file sizes have doubled since 1984 [39]. The largest file was 2 GB, which is about 4,000 times bigger than the largest file in

1984.

Data Grid

In [33], Iamnitchi and Ripeanu studied the characteristics of the DZero data-intensive physics project. They observed that the file size distributions and file popularity distributions did not follow traditional models. The file popularity did not follow Zipf-like distribution. The reason for this seems to be the nature of the physics events which are recorded. All the events seems to be equally popular. The file size distribution did not follow a heavy-tailed distribution. The file sizes varied from a few KB to 1.9 GB. File size distributions had 2 different peaks at 20 MB and 255 MB.

P2P Systems

In [52], the file popularity in Gnutella was studied. It was found that the very popular files were equally popular. The popularity distribution of rest of the files followed Zipf-like distribution.

Recent studies in peer-to-peer file-sharing applications such as Gnutella, Kazaa and Napster confirm that different file size distributions emerge with different content types (predominantly multimedia in this case) [50].

Overall, most of the file size distributions observed are log-normal with a heavy tail. A large number of files are small. File popularity in web requests follow Zipf-like distribution. There are a lot of files that are less popular. About 20% of the files have lifetimes as long as 5 months.

3.2 Data Grouping

In [10], Amer et al. create groups of files that are accessed together based on file access patterns and use these groups to prefetch files. They use an aggregating cache that maintains a successor list of files for each file that is accessed. The sequence in which files are requested is used to identify the successor of a file (next file requested). For each new file access, these lists are traversed until a unique path is identified. Once this unique path

is identified, the rest of the files in this path are prefetched. They observed a 20 to 1,200% improvement in cache hit rates. Filecules are different from the groups identied in [10] in that filecules are disjoint sets of files and grouping is not based on the order in which the files are accessed.

In [53] Tait and Duchamp analyze the use of file working sets for improving cache performance using prefetching. Their algorithm builds distinct working trees based on file access sequence and patterns. For every job, they track the file access sequence and compare it with the existing working trees. Prefetching is delayed until the sequence matches only one working tree. When a unique working tree is identified, the remaining files of that working tree are prefetched. Their experiments with file access traces from a SunOS machine prove that LRU with prefetching outperforms conventional LRU.

In [30], Gkantsidis et al. analyzed grouping of files in Windows Update—a software update service. They experimented with clustering of files using cosine correlation between pair of files. The cosine correlation determines the probability of two files being requested together. The cosine correlation of a pair of files is 1, if and only if both the files are always requested together. The threshold correlation used for grouping was 0.9. They observed that 98% of the files formed 26 non-overlapping groups. 5 largest groups accounted for 97% of the total software update requests. They also tried clustering the update patches but observed that they did not cluster as much as files. Filecules are defined analogously but independently of this work [30], but we do not group files that have correlation coefficients smaller than 1.

Ganger and Kaashoek [28] use explicit grouping in which files that are used one after the other are placed in adjacent locations on the disk and accessed as a whole group. Griffioen and Appleton [31] consider two files related (and thus, part of the same group) if they are opened within a specified number of file open operations from each other.

3.3 Stack Depth Analysis

In [12], Arlitt and Jin study the workload of the 1998 world cup web site. They use stack depth analysis to show the temporal locality in the workload. They describe that

if the average or median stack depth is relatively small compared to the maximum stack depth in the workload (total number of files), then there is good temporal locality and vice versa. The 90^{th} percentile of the stack depth accessed was about 4% of the maximum stack depth. This shows that their workload has good temporal locality.

In [6], Acharya, Smith and Parnes characterize videos accessed on the web. They plot the percentage of stack depth accessed during Least Recently Used replacement algorithm. The plot shows that most of the stack depth accessed is a small value (70% of the stack depth accessed is less than 10) and hence good temporal locality in the video files accessed.

We use this stack depth analysis in our studies to verify that DZero workload traces exhibit good temporal locality (Section 5.1) and hence algorithms that take advantage of temporal locality should be utilized to process data in such systems.

3.4 Caching in Web

In [46] and [48], Pitkow and Recker propose a caching algorithm that adapts to document hit rates and user access patterns. Based on models from psychological research on human memory, they identified that recency rates of document access history can be used to predict future document access. They used web workloads from Georgia Institute of Technology. They calculated the probability of access of a file based on recency of use in the access history and frequency of access within a given history window. Cache replacement was performed based on this calculated probability and was shown that the probability of a cache miss using this prediction was less than 0.1. This shows that prediction mechanisms using recency and frequency of data access with recency having more weightage predicts future document access most of the time (probability of correct predition is 0.9).

In [8], Aggarwal and Yu propose a modified version of LRU that evicts a group of documents with least dynamic frequency. Dynamic frequency of a group of documents is the sum of the inverse of difference between current time and last access time of each document, and hence account for the recency of use. The group of files with the Least dynamic frequency is evicted from the cache. They show that this algorithm prevents caching of rarely used objects and also reduces fragmentation.

In Akamai-Content Delivery Networks [36], web content delivery is accelerated by prefetching data. When a client requests for a web page, it is sent to the edge server, a caching server located closer to the user. The edge server sends the request to the appropriate web server for content. Once the edge server receives the content, it is parsed to identify embedded objects. The edge servers send content to the browser and simultaneously send requests to prefetch the embedded objects. The content is resolved by the browser and it sends requests to the edge server to load the embedded objects. Since the edge server has prefetched the embedded objects, the loading of this data is accelerated. In this case, the entire data that is prefetched is utilized by the client and the data that is prefetched depends on the contents of the client request i.e., objects embedded in the client request. In case of scientific data, such explicit relationships do not exist between files. We predict this relationship based on usage data from the past.

3.5 Caching in Data Grids

In [41], Otoo et al., propose a new disk cache replacement policy for SRM [51] in data grids, Least Cost Beneficial (LCB-K) replacement policy based on at most K backward references. They use workload from JasMINE (Jefferson Lab Asynchronous Storage Manager) [4] for a period of 6 months. According to their LCB policy, a utility value is calculated for each file that is not currently in use by an running job. Files are evicted from cache in non-decreasing order of their utility values. Their algorithm is compared with LFU, LRU, LRU-K [40], Greedy Dual Size [22] and MITK (a variant of LRU-K). They measure average cost per reference as total cost in time units divided by the total number of references to the file. LCB-K is shown to provide the lowest average cost per reference. In [45], LCB-K is shown to perform better than the other caching algorithms on workloads from the National Energy Research Scientific Computing facility.

In [42], Otoo, Roten and Romosan propose a optimal file-bundle caching replacement algorithm, Greedy Relative Value (GRV) that reduces the volume of data transfer and increases the throughput. This algorithm determines the optimal set of files that needs to be loaded into the space available in the cache such that the throughput of the system

can be improved. The relative value of a request is a function of the popularity of the request and the adjusted size of the files requested. The adjusted size of a file is the size of the file divided by the popularity of the file. Files are loaded into the cache based on the relative value of the requests. The GRV algorithm is explained in detail in Section 5.2.3. The performance of this algorithm is compared with the Landlord algorithm proposed in [22] and [57].

In [44], Otoo, Rotem and Seshadri use the same relative value presented above to also change the order in which the jobs in the waiting queue are processed. The jobs with the highest relative value is processed first. This ensures that the jobs that can utilize the current set of files in the cache can be processed immediately. They compare their algorithm with First-Come First-Served job scheduling policy. The results show that the average response time and average queue length for GRV is smaller than LRU cache replacement policy with FCFS scheduling. In [43], Otoo et al., compare GRV with Greedy-Dual Size.

Since the DZero traces exhibit good temporal locality and also because GRV provides the capability of running jobs that utilize the current state of the cache itself, we propose a combination of the Least Recently Used cache replacement algorithm with job scheduling based on the relative value mentioned in GRV algorithm (Section 5.2.3).

Filecules are different from the file-bundles used in [42], [44] and [43]. File-bundles consist of the entire set of files used by a job, also known as a "collection". Hence, a file-bundle might consist of one or more filecules.

CHAPTER 4

FILECULES AND THEIR CHARACTERISITICS

A typical job in DZero requests multiple files (Section 2.4). Analysis of these requests reveals that the files requested are often correlated. Such a group of correlated files is a filecule.

In this chapter, we formally define a filecule and we characterize filecules in terms of size, popularity and lifetime. These properties will be compared with those of data in traditional file systems and the Internet. We observe that the properties of filecules follow different distributions than those observed in traditional file systems.

Inspired from the definition of a molecule, we define a filecule as an aggregate of one or more files in a definite arrangement held together by special forces related to their usage. We thus consider a filecule as the smallest unit of data that still retains its usage properties. We allow one-file filecules as the equivalent of a monatomic molecule, (i.e., a single-atom as found in noble gases) in order to maintain a single unit of data (instead of multiple-file filecules and single files).

Formally, a set of files $F_1, ..., F_n$ form a filecule G if and only if $\forall F_i, F_j \in G$ and $\forall G'$ such that $F_i \in G'$, then $F_j \in G'$. Properties that result directly from this definition are:

1. Any two filecules are disjoint.

2. A filecule has at least one file.

3. The number of requests for a file is identical with the number of requests for the filecule that includes that file. Thus, popularity distribution on files and filecules is the same.

Table 4.1 shows the statistics of size, popularity and lifetime for files and filecules.

Table 4.1 Statistics of Size, Popularity and Lifetime

Property	Minimum	Maximum	Mean	Median	Standard Deviation
File size	234 bytes	1.98 GB	0.3859 GB	0.3773 GB	0.3230 GB
Filecule size	23 KB	16,051 GB	3.9859 GB	0.9419 GB	54.5137 GB
File popularity	1	996	12	3	25
Filecule popularity	1	996	41	30	50
File lifetime	15 secs	27 months	4 months	1 month	5 months
Filecule lifetime	15 secs	27 months	8 months	7 months	5 months

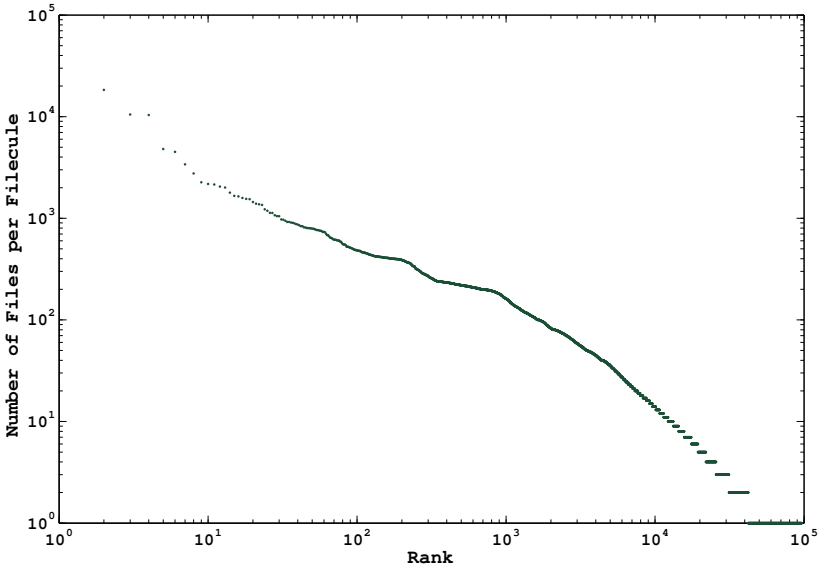

Figure 4.1 Number of Files Per Filecule

Figure 4.1 shows the distribution of the number of files per filecule. The figure shows that there are a few large groups of files (5% of filecules above 15 TB) and many small groups of files (56% are one-file filecules).

4.1 Size Characteristics

Analysis of file and filecule size characteristics of the DZero Experiment provides an understanding of the typical data set size used by physicists. We intend to see how the size distribution changes due to grouping of data into filecules. This characterization is useful for modeling data size when considering filecules for data management.

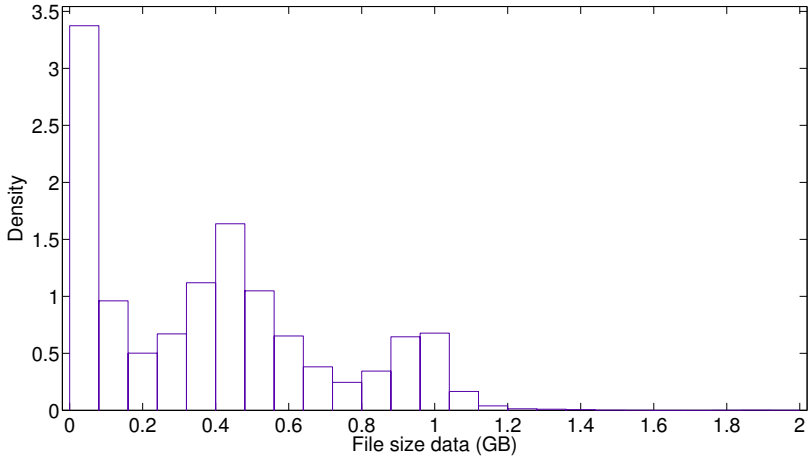

Figure 4.2 File Size Distribution

4.1.1 File Size

Figures 4.2 and 4.3 show the file size distribution and file size vs. rank plot. The file with the largest size has rank 1. The smallest file is 234 bytes and the largest size is 2.1 GB. The mean and median files are 0.4 GB. Since the mean and median values are same, the files are equally distributed along the mean (Number of files larger than mean $\simeq$ Number of files smaller than mean) It can be noticed in Figure 4.2 that there are many files (69%) that are smaller than 110 MB. The next popular file sizes are 440 MB to 550 MB and 1 GB to 1.1 GB. Further analysis of these popular file sizes shows that most of the files fall in the category of files smaller than 10 MB, between 450 MB and 470 MB and between 1.03 GB and 1.04 GB. The distribution follows a similar pattern observed in [33] except that the peak file sizes are different. In [33], there were fewer files with file sizes that represent peaks in our study. This could be due to shorter traces in [33] (6 months in [33] compared to 2+ years in our study). The multiple peaks observed in the file size distribution is due to the different peaks observed in different data tiers as shown in Figure 4.4. The files in different data tiers are generated as a result of various reconstruction applications (Section 2.2) which can be attributed to different locations of peaks in Figure 4.4.

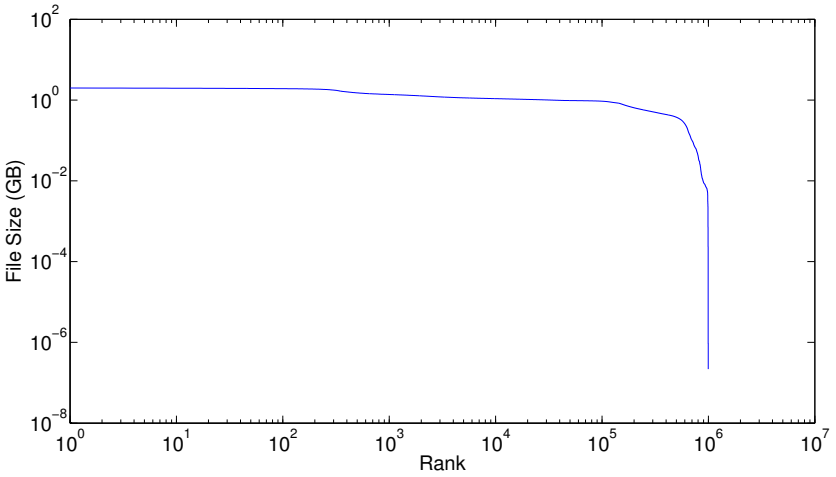

Figure 4.3 File Size vs. Rank

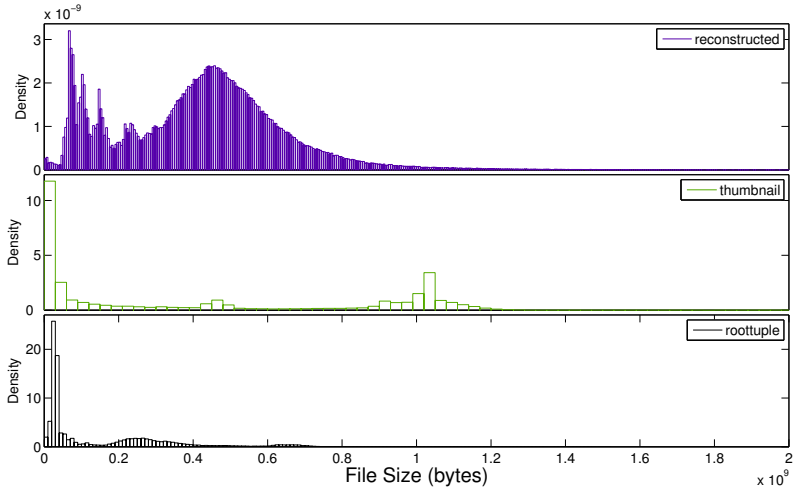

Figure 4.4 File Size Distribution Per Data Tier

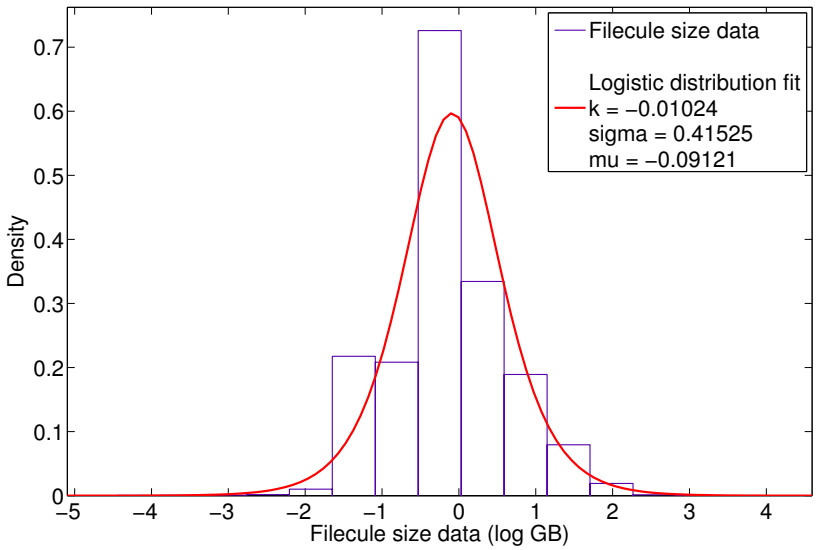

Figure 4.5 Filecule Size Distribution

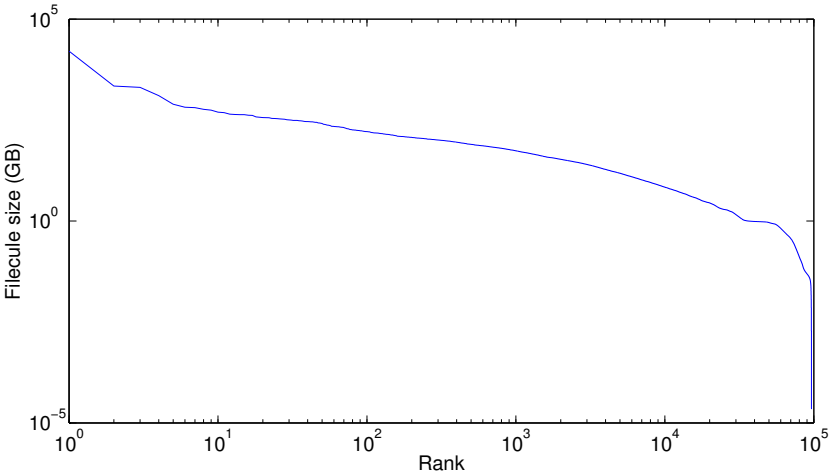

Figure 4.6 Filecule Sizes in Decreasing Order

4.1.2 Filecule Size

Figures 4.5 and 4.6 show the filecule size distribution and filecule size ordered in decreasing order of size. The largest filecule is 15.7 TB. The smallest filecule is 23.5 KB. All the files (2,264 files) with size less than 23.5 KB have been grouped into filecules. The largest filecule size is about 8,000 times the largest file size. This filecule has 18,326 files. The mean filecule size is 4.2 GB and the median filecule size is found to be 1.1 GB. This positive skewness is because of the influence of few large filecules (5% have a size larger than 15 TB). Log-logistic distribution with parameters mentioned in Figure 4.5 best fits filecule size distribution. This contradicts the log-normal size distribution of data observed in [25] and [17]. The difference between log-normal and log-logistic distribution is that the log-logistic has a fat tail (larger number of large files). The curve of a log-logistic distribution increases geometrically with small values, flattens in the middle and decreases slowly at high values.

4.2 Popularity Characteristics

Popularity of file or filecule is measured as the number of times file or filecule has been requested. Popularity distributions shows patterns in data usage. We intend to see if usage patterns change with grouping data in filecules. This characteristic is particularly relevant for predicting caching performance when using filecules as the data abstraction.

4.2.1 File Popularity

Figures 4.7 and 4.8 show the file popularity distribution and file popularity in decreasing order. The most popular file was requested by 996 jobs (34 unique users). About 30% of the files have been used by only one job (file popularity =1). The rank for the median file popularity (3 jobs) is 700,000, i.e., about 30% of the total number of files (997,227) are requested only by one or two jobs. 6.5% of the total number of files (65,536 files ordered by file popularity) account for 45% (5,247,549 requests) of total requests (11,568,086 requests). This is similar to the observation in [15] where a small set of files account for majority of the requests. Only 4 files are highly popular with a file popularity of 996. The popularity

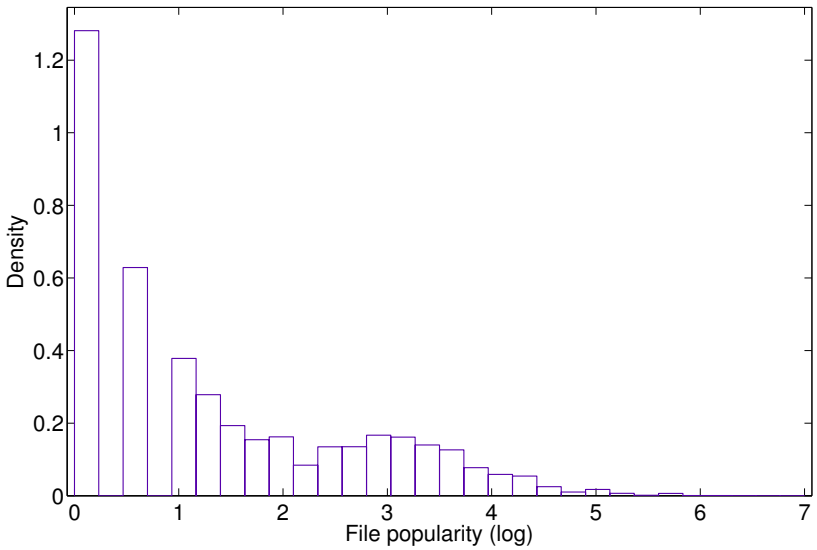

Figure 4.7 File Popularity Distribution

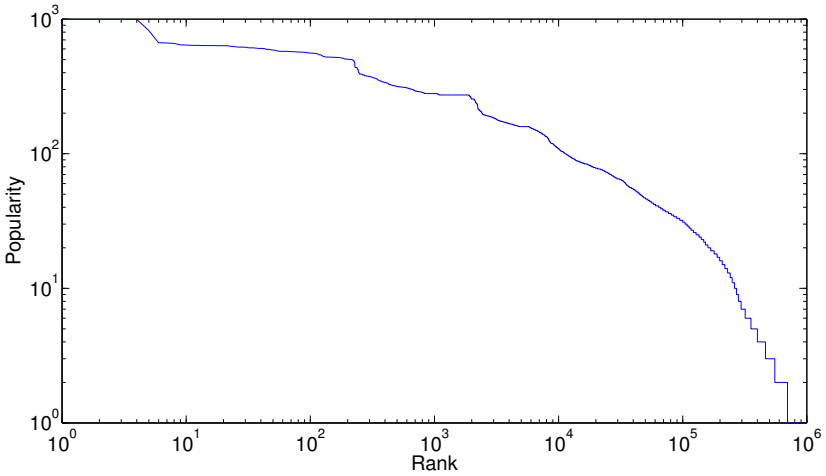

Figure 4.8 File Popularity vs. Rank

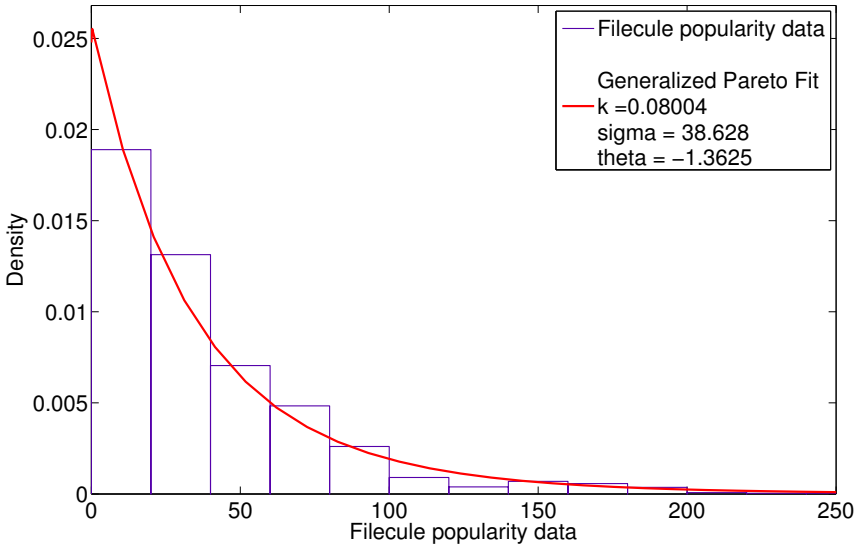

Figure 4.9 Filecule Popularity Distribution

distribution is heavy tailed (Figure 4.8) similar to observations in [33] and [11]. Also, the distribution does not follow the Zipf observed in [52], [9], [24] and [11].

4.2.2 Filecule Popularity

Figures 4.9 and 4.10 show the filecule popularity distribution and filecule popularity vs. rank plot. There is only one filecule with maximum popularity observed. There are a total number of 3,918,553 filecule requests for 96,454 unique filecules. 49.7% (47,910) of the filecules account for 86% (3,381,638) of the requests. 6.5% (6,270) of the filecules account for 33.3% (1,126,255) requests. The generalized Pareto distribution best fits the filecule popularity data. The parameters are as given in Figure 4.9.

Figure 4.10 clearly shows that the popularity distribution does not follow a Zipf's law that is observed in [9], [24] and [11]. The filecule popularity is not as heavy tailed as file popularity. This is because files with less popularity group better into filecules than the very popular files. This is similar to the discussion in [10], which mentions that there might be a few more popular files which will be used along with a lot of different file sets. Hence when trying to identify disjoint groups of files, these files remain single rather than form groups.

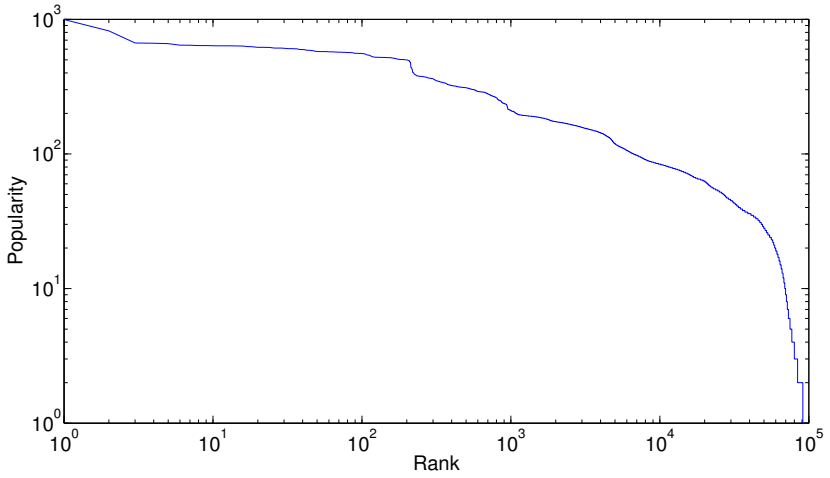

Figure 4.10 Filecule Popularity vs. Rank

4.3 Lifetime Characteristics

We define the lifetime of a file or filecule as the time difference between the start time of the first job that accessed the file and the end time of the last job that accessed it in our trace. This is defined as the active period of the data.

4.3.1 File Lifetime

Figures 4.11 and 4.12 show the file lifetime distribution and file lifetime in decreasing order of lifetime. 40% (396,341) of the files have lifetime shorter than one week. Median file lifetime is 712 hours ($\doteq$1 month). About 35% of the files have a lifetime greater than or equal to 5 months. This is similar to the observation in a web proxy workload [11], where 20% of the files were active after 5 months. 294,355 (30%) files have a lifetime of less than or equal to a day (24 hours). This number is less than the ones mentioned in [11] which reports 50% of inactive files on the next day. This indicates that the files in DZero have longer lifetimes than those observed in the Internet. This can influence the effectiveness of caching. On an average 4.54 TB of distinct data is requested each day. 30% of this data becomes useless on the next day and 70% of the data is still active. In order for a cache to be effective, it needs to retain around 70% of the data from the previous day. This shows

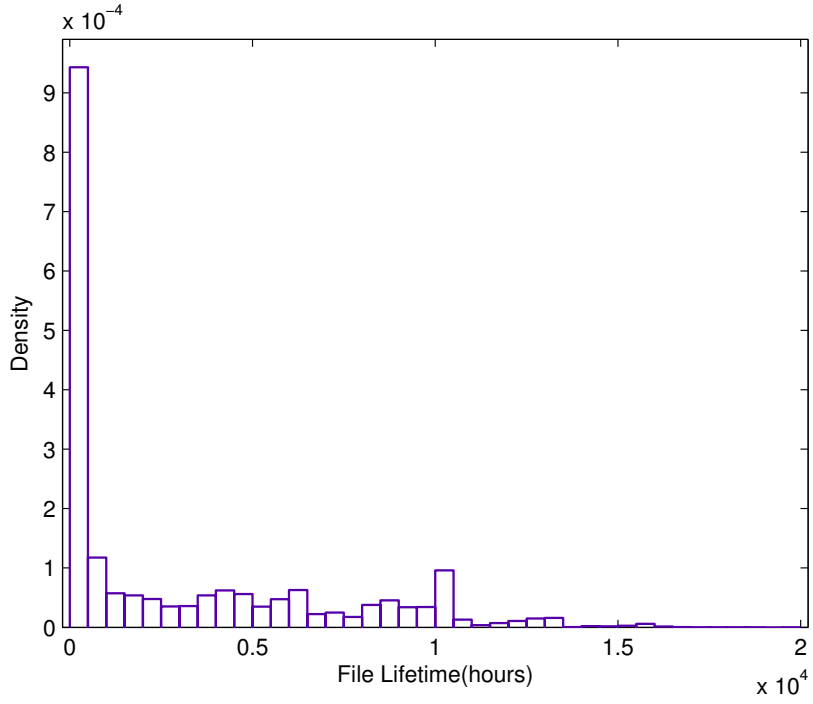

Figure 4.11 File Lifetime Distribution

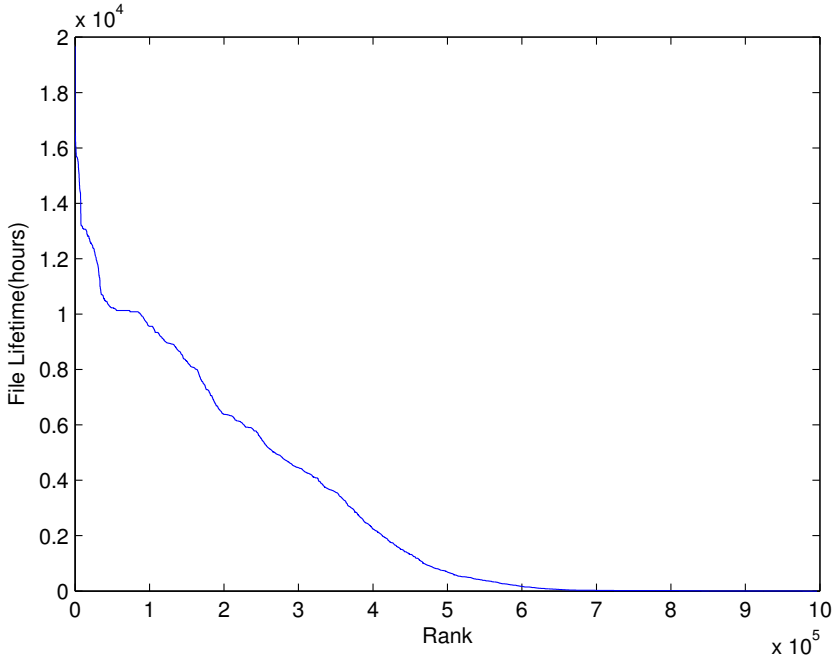

Figure 4.12 File Lifetime vs. Rank

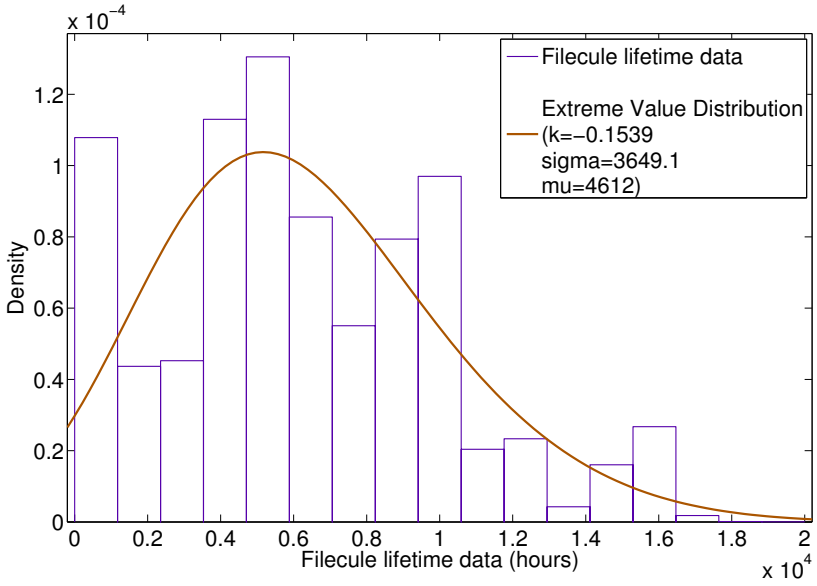

Figure 4.13 Filecule Lifetime Distribution

us that any cache size that is less than 3.5 TB (70% of 5 TB) will not be enough to take advantage of the temporal locality in the data.

4.3.2 Filecule Lifetime

Figures 4.13 and 4.14 show the filecule lifetime distribution and filecule lifetime in decreasing order of lifetime. The best distribution fit for the data is extreme value distribution distribution with shape parameter (k)=–0.1539, scale parameter (sigma)=3,649.1 and location parameter (μ)=4,612. This is different from the hyperexponential distribution observed in [25] Windows file system. More than 70% of the filecules are active after 5 months. This 70% of filecules is equal to 35% of files (See Section 4.3.1). This also shows that more popular files have not grouped well with other files which is similar to the observation in Section 4.2.2. 5,175 (5%) filecules become inactive after a day. This 5% of the filecules account to 30% of the files which become inactive after a day. This suggests that files with short active periods have a better tendency to group than files that have long active periods, similar to observation in Section 4.2.2, which can be due to a file being accessed along with different file sets by different jobs.

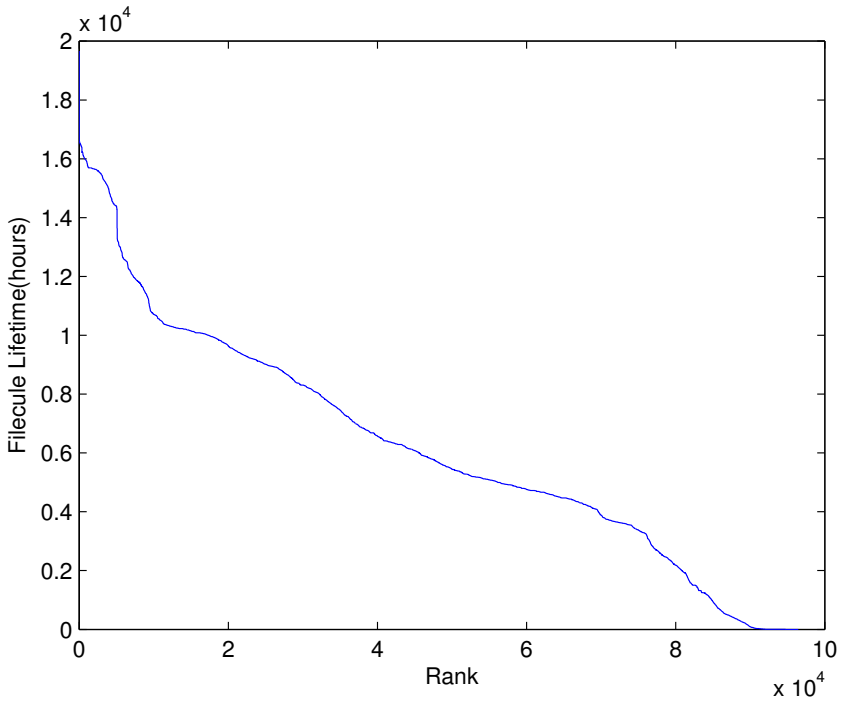

Figure 4.14 Filecule Lifetime vs. Rank

Table 4.2 Coefficients of Correlation for File Properties

File size & File popularity	0.1235
File size & File lifetime	-0.0172
File popularity & File lifetime	0.3888

4.4 Correlation Between Size, Popularity and Lifetime

The coefficients of correlation between file attributes are given in Table 4.2 and those of filecule are shown in Table 4.3. There is no correlation between these characteristics which indicates that using data size or lifetime to make decisions about data popularity is incorrect. For example, evicting data from the cache based on size or time elapsed since first access will not improve the performance of the cache.

Table 4.3 Coefficients of Correlation for Filecule Properties

Filecule size & Filecule popularity	-0.0390
Filecule size & Filecule lifetime	-0.0602
Filecule popularity & Filecule lifetime	0.3989

4.5 Impact on Resource Management

Filecules can be used as a new granularity for resource management in scientific grids. The nature of jobs in such communities, i.e., jobs requesting multiple files, requires consideration for correlation between files. This correlation is used to group files into filecules. Grouping related files into filecules reduces the number of objects that need to be managed, and naturally preserves locality of use.

In Chapter 5, we showed that using filecules to prefetch data into the cache improves the byte hit rate. But filecules can be also applied to data replication, job scheduling, resource selection, and data staging.

By using filecules for data replication, related data can be stored together at the same location. This ensures faster data search and retrieval. Moreover, it can guide job scheduling for selecting a computational resource close to where the data needed by the job is stored.

The degree of correlation between filecules can be utilized for data staging. A threshold of correlation can be used to determine how far away two filecules should be stored. If the threshold of correlation is met, the filecules are stored in nearby locations.

Instead of identifying filecules in one central location, they can be identified locally in multiple storage locations. This will enable the system to identify filecules that define the usage patterns local to that storage. This can improve the quality of resource management.

4.6 Summary

The filecule size distribution follows Log-logistic distribution indicating that the decrease in the number of large files is not as steep as it is observed in Log-normal distributions identified in Windows file systems [25] and web client traces [15]. The filecule popularity distribution is a Generalized Pareto distribution. The file and filecule lifetime distributions indicate that the data in DZero have longer lifetimes than data observed in web proxy workloads [11]. The correlation coefficients show that data prediction or eviction decisions in storage cannot be made based on data size and data lifetime.

CHAPTER 5

CACHING

Storage Resource Managers (SRM) [51] and Storage Resource Brokers (SRB) [47] provide caching and data storage services for data-grids. Data that needs to be processed by a job is located on a Mass Storage System (MSS) that can be located locally or remotely. SRM has a large capacity disk cache and this cache space is utilized to store data that is read from MSS. SRMs facilitize high data availability by staging data and by masking any failures in data transfer. This chapter discusses caching algorithms for such SRMs.

Data can be loaded into SRM disk cache on a demand basis or can be pre-staged based on a usage prediction method. Data is loaded in the cache on a demand basis when a job requests for data and the data is not available in the cache. Prediction methods are used to predict what data might be required by a job in the future. Prediction is typically done based on the history of data usage [42, 44, 43, 10].

In this chapter, using stack depth analysis, we show that the DZero workload has good temporal locality. We compare the Least Recently Used (LRU) cache replacement algorithm and the Greedy Request Value (GRV) cache replacement algorithm. We show that using filecules for data prefetching in the cache provides better prediction of data usage and hence better byte hit rate. We identify the drawbacks of using a First-Come First-Served (FCFS) job scheduling algorithm and compare FCFS with job scheduling algorithm in GRV algorithm. We show that a combination of caching using filecules and job scheduling using GRV algorithm provide good byte hit rates and short job waiting times.

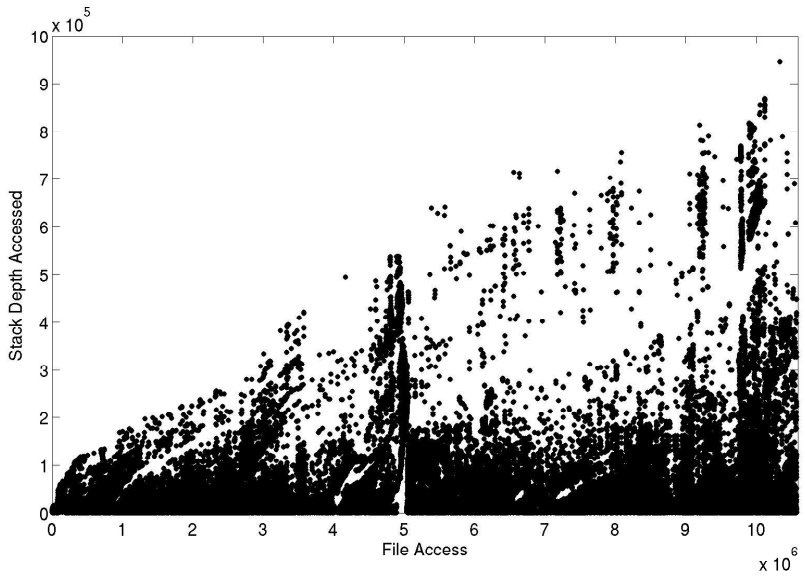

Figure 5.1 Stack Depth Analysis of File Requests

Table 5.1 Stack Depth Analysis - Statistics

Measure	Value
Maximum	946,600
1 percentile	85
10 percentile	960
50 percentile (Median)	12,260
90 percentile	90,444
Standard Deviation	79,300

5.1 Stack Depth Analysis

Stack depth analysis [12, 6] is a method that measures the temporal locality of a workload is tested. A stack is used to represent a cache. When a job requests a file, the most recent location of the file in the stack is identified. The depth of this location in the stack from the top of the stack is the stack depth for the file access. The stack depth accessed for the entire workload is measured.

There are approximately 11.5 million file accesses in the workload. From Figure 5.1, it can be seen that all stack depths are less than 1 million. This is less than 10% of the total number of file accesses. Table 5.1 shows the statistics of the plot.

Table 5.2 Average Number of Files in Cache

Cache size (TB)	# of files	% of access depth greater than column 2
50	132,830	6.15
25	66,415	13.84
10	26,566	31.83
5	13,283	48.40
1	2,656	76.80

The strength of the temporal locality determines the stack depths accessed: the stronger the temporal locality, the smaller the stack depth. As found in Figure 5.1, there is a thick band closer to the x axis which indicates that there are a large number of small stack depths. The plot becomes sparse as only a small number of large stack depths are accessed.

Table 5.2 shows the number of files that can be accomodated in each cache size and the percentage of stack depth accesses that are greater than the number of files that can be accommodated. The DZero workload contain 996,227 files whose sizes add up to approximately 375 TB. We obtained the fraction of the number of bytes that can be accommodated in each cache size (cache size/375 TB), and used this fraction to calculate the number of files that can be accomodated in the cache. The number of files that can be accommodated for all cache sizes, with the exception of 1 TB, is less than the median stack depth accessed.

The average number of unique files accessed per month is 100,201. Figure 5.1 shows that most of the stack depth is less than 1 x 10^5 and the 90th percentile is 90,444. This indicates that most of the files requested have been accessed previously within a window of one month.

5.2 Cache Replacement and Job Scheduling Algorithms

Any storage has limited capacity. The amount of data that needs to be stored in a system increases with time and thus it reaches a point where there is no more space left in storage to add new data. Under such circumstances, the system needs to make decisions about what data needs to be retained and what can be evicted from storage. These decisions are made based on different parameters like age of the data, size of the

data, etc. SRMs also have limited disk cache. When the disk cache is full, SRMs evict data to make space for new data. SRMs also schedule jobs to improve the throughput of the system. This Section describes the various cache replacement and job scheduling algorithms used in our experiments.

The following are the various combinations of cache replacement and job scheduling algorithms used:

1. Least Recently Used cache replacement using files with First-Come First-Served job scheduling algorithm with infinite queue length (File LRU)

2. Least Recently Used cache replacement using filecules with First-Come First-Served job scheduling algorithm with infinite queue length (Filecule LRU)

3. Greedy Request Value with infinite queue length (GRV)

4. Greedy Request Value with threshold queue length of 1,000 (GRV-1000)

5. Least Recently Used cache replacement using files with scheduling using Greedy Request Value with infinite queue length (LRU-Bundle)

6. Least Recently Used cache replacement using files with scheduling using Greedy Request Value with threshold queue length of 1,000 (LRU-Bundle-1000)

5.2.1 Least Recently Used Cache Replacement Algorithm

Least Recently Used (LRU) [5, 13, 14] is the cache replacement algorithm used currently at the Fermi National Accelerator Laboratory. LRU is a temporal locality-based algorithm. According to this algorithm, when storage is full and a new data object needs to be loaded into the storage, the replacement algorithm chooses the least recently used data in the storage to be evicted to make space for the new data object.

5.2.2 First-Come First-Served Job Sceduling Algorithm

The First-Come First-Served (FCFS) job scheduling algorithm uses the principle of a queue to schedule jobs to run. The job that is first submitted to the system is scheduled

to run before any other job that is submitted later. There is no bias in the order in which jobs are run. If space is not available in the cache to load files for the job that is ahead of the queue, all jobs wait in the queue until the first job starts to run.

5.2.3 Greedy Request Value Algorithm

Otoo et al., [42, 44, 43] proposed the Greedy Request Value (GRV) algorithm which combines cache replacement with job scheduling. The goal of this algorithm is to reduce the amount of data transferred into the cache for a job to run and increase the throughput of the system by utilizing the existing contents of the cache. A collection is defined as the set of files requested by a job. Each file requested is assigned a relative value based on it's size and popularity (Equation 5.1). Each job is assigned a relative value based on the popularity of the collection of files requested and the relative value of the files (Equation 5.2). Jobs are scheduled according to this value: the job with the largest value is scheduled first.

$$v_{f_i} = \frac{s(f_i)}{n(f_i)} \tag{5.1}$$

where v_{f_i} is the relative value of a file f_i, $s(f_i)$ is the size of file f_i and $n(f_i)$ is the popularity of file f_i.

$$V_r = \frac{n(r)}{\sum_{i=1}^{N} v_{f_i}} \tag{5.2}$$

where V_r is the relative value of request r and $n(r)$ is the popularity of the request r.

When a new job is submitted to the system, this job enters the waiting queue. The relative value of this job is calculated based on Equations 5.1 and 5.2. The job with the largest relative value is scheduled as the next job to run.

When a job is scheduled to run, the space required for the job is estimated. The space required is the difference between the sum of the size of files in the collection and the sum of size of files for the request that are already available in the cache. The algorithm calculates the space available in the cache. Space available in the cache is the sum of the free space

in the cache and sum of the size of files that are not used by any currently running job. If the space available is greater than or equal to the space required, the job can be processed.

When a job can be processed, the space required for the request is reserved and the remaining space (the difference between the space available and space required) is used for prefetching data. Data is prefetched based on the relative value of the requests in the history. The files of the request with the largest relative value in the history is loaded in the available space.

5.2.4 Queue Freezing

Queue freezing is a method in which the order in which the jobs will be scheduled is decided once and that order is followed until all the jobs in the frozen queue have started to run. Any new job that arrives at the queue needs to wait until all the jobs in the frozen queue have started to run. This method is used to avoid thrashing of jobs due to job scheduling algorithms.

Since FCFS is an unbiased algorithm and the order in which jobs are processed is never changed, there is no queue freezing for FCFS. In GRV and LRU-Bundle, certain jobs can suffer long delays because of very small request relative values. In order to avoid these long delays, queue freezing is implemented for GRV and LRU-Bundle. We chose 1,000 jobs as the threshold queue length for queue freezing because it is the average number of jobs submitted per week. Queue freezing also takes advantage of the temporal locality characteristic in the workload.

For GRV and LRU-Bundle, when the queue length reaches this threshold, the relative values of all the jobs in the queue is calculated and the jobs are sorted in decreasing order of their relative values. The queue is frozen with these jobs in their scheduled order. The relative value of any incoming new job is not computed until all the jobs in the frozen section of the queue are processed. This also has an effect on the computational overhead for scheduling.

Once all the jobs in the frozen section of the job waiting queue are processed, the relative value of all the jobs in the remaining job waiting queue are computed. If again the queue length is greater than or equal to threshold, the first 1,000 jobs are sorted in

decreasing order of their relative values and the queue is frozen. If the queue length is less than the threshold, the jobs are processed without freezing the queue.

5.3 Metrics

The goal of our experiments is to identify a combination of cache replacement and job scheduling algorithm that will utilize the temporal locality in the workload, reduce the volume of data transfer into the cache and avoid job thrashing with small computational overhead. We identified metrics that will quantify our goals for the algorithms mentioned in Section 5.2. The traditional metric used in evaluating cache replacement algorithms [42] that use predictive prefetching is byte miss ratio. Queue wait time measured in terms of number of iterations a job remains in the queue [43], average response time and average queue length [44] are the metrics used to compare the performance of job scheduling algorithms. Response time in [43] is measured as the time difference between the arrival time of the job and the time when the file requests are completed.

Byte Hit Rate

Byte hit rate is the most commonly measured performance metric for cache replacement algorithms. Byte hit rates indicate the percentage utilization of the content of the cache.

$$ByteHitRate = 100 * \frac{ByteHit}{ByteHit + ByteMiss} \tag{5.3}$$

For our experiments, we measure byte hit rate per job. Since data is prefetched into the system, we intend to quantify the utility of this prefetching for the job that is run. This does not quantify how much data is moved into the cache for this job to be processed.

Percentage of Cache Change

The percentage of cache change is a measure of the amount of data loaded into the cache in order to run a job. This is measured as the percentage difference in the bytes in cache before and after the cache is loaded with files required by a job. This measure is an

indicator of the volume of data transfer that occurs in order to process a job. The formula used for calculation is as below:

$$VC_{i-1,i} = \frac{S(C) - (M_{i-1,i} + min(F(C_{i-1}), F(C_i)))}{S(C)} * 100 \tag{5.4}$$

where C_i is the content of the cache after loading the files necessary for the i^{th} job, $VC_{i-1,i}$ is the percentage of cache change due to processing of the i^{th} job, $S(C)$ is the size of the cache, $F(C_{i-1})$ is the free space in the cache before the files for the i^{th} job are loaded, $F(C_i)$ is the free space in the cache after the files for the i^{th} job are loaded, $M_{i-1,i}$ is the size of the files in $C_{i-1} \cap C_i$.

Queue Length

The queue length indicates how many jobs are in the waiting queue. This metric shows the efficiency of the scheduling algorithm in terms of utilizing the available cache contents and the free space to schedule new jobs.

Job Waiting Time

Job waiting time indicates how long a job was retained in the waiting queue before it was scheduled for processing. This measure shows if some of the jobs experienced very long delays due to the scheduling algorithm.

Scheduling Overhead

Scheduling algorithms do some computations based on various attributes of the jobs in the waiting queue and make scheduling decisions based on the results of those computations. These computations are overhead to the system. The computational overhead can be represented as a function of the number of computations performed to make a decision. This value is an indicator of how many CPU cycles are utilized to schedule the next job. For FCFS job scheduling, the scheduling overhead is always zero. For GRV and LRU-Bundle, scheduling overhead is dependent on the number of jobs in the waiting queue.

5.4 Experiment Setup

Our experiments consisted of simulation of disk cache using real workloads from the DZero Experiment at Fermi National Accelerator Laboratory. We implemented the algorithms listed in Section 5.2 and compared them using the metrics listed in Section 5.3. The GRV implementation is based on the OptCacheSelect and OptFileBundle algorithms described in [43]. LRU-Bundle uses least recently used cache replacement algorithm. GRV and LRU-Bundle schedule jobs based on the relative request value of the job calculated as given in Equation 5.2.

The workload obtained from the DZero Experiment resides in a MySql database. The workload consists of two sets of information: information about the start time and end time of each job and information about the list of files requested by each job. The workload has data about all jobs submitted from January 2003 to March 2005. The simulator is a Java program consisting of 3,500 lines of code which connects to the MySql database using JDBC to obtain details about each job. The input values to the simulator are the size of the cache, the cache replacement algorithm and the job scheduling algorithm. Our simulations were run on the entire workload. A job run and, in turn, file caching can be triggered by two events: a job arrival or a job completion. When the total size of a job, i.e. the sum of the size of files requested by a job is bigger than the size of the cache, the job is ignored. The metrics listed in Section 5.3 are reported after each job is scheduled to run.

Simulations were run for 6 different caching algorithms with 5 different cache sizes leading to 30 different runs. For calculating the optimal set of files to be loaded into the cache for GRV algorithm, the history of jobs from the previous 1 week was used. The average run time for each simulation is around 6 hours on a Pentium II with 2 GB memory running Linux operating system.

5.5 Experimental Results

Our experiments aim to identify the impact of these algorithms over a range of cache sizes (1 TB to 50 TB). Our goal is to identify a set of algorithms for the particle physics

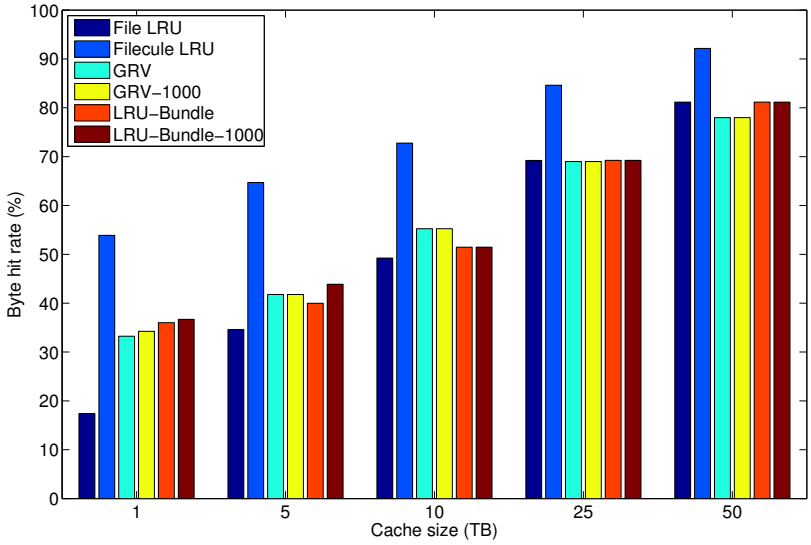

Figure 5.2 Average Byte Hit Rate

science grid that will improve the throughput of the system with least amount of data movement. We intend to see which algorithms are suitable for small and large cache sizes.

The queue length for GRV does not exceed the threshold value of 1,000 for cache sizes of 5 TB and above. Hence, GRV and GRV-1000 algorithms are the same for cache sizes of 5 TB and above. Similarly, the queue lengths for LRU-Bundle and LRU-Bundle-1000 algorithms do not exceed threshold value for cache sizes of 10 TB and above, and hence LRU-Bundle and LRU-Bundle-1000 are the same for these cache sizes. At 50 TB, all jobs are scheduled to run as soon as they are submitted to the system. Hence File LRU and LRU-Bundle run the same for 50 TB cache size.

5.5.1 Byte Hit Rate

Figure 5.2 shows the variation of byte hit rate for all the algorithms with increasing cache size. Filecule LRU has highest byte hit rate for all cache sizes. This is due to the effective prefetching of filecules. This is an upper bound for prefetching using filecules because the filecule definitions used are optimal. We used the entire workload to identify these filecules. In Chapter 6,we discuss the effect of using window of history to identify filecules.

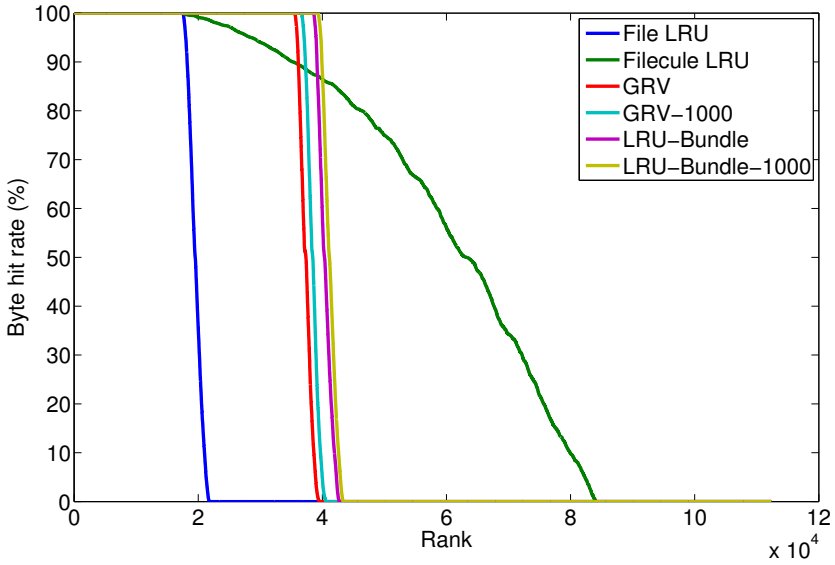

Figure 5.3 Byte Hit Rate for Cache Size of 1 TB

For cache sizes up to 10 TB (Figures 5.3, 5.4 and 5.5), File LRU has lower byte hit rate than GRV and LRU-Bundle. GRV and LRU-Bundle take advantage of the contents of the cache by changing the order in which jobs are processed. They schedule those jobs that can utilize the contents of the cache rather than on a FCFS basis.

For larger cache sizes of 25 TB and 50 TB (Figures 5.6 and 5.7), File LRU hash higher byte hit rate than GRV and shows similar performance as LRU-Bundle. This is because for these cache sizes, the number of files that can be accommodated in the cache is high enough that the files required for the jobs are found in the cache. For 50 TB and 25 TB, only 6.15% and 13.84% of the stack depths are higher than the average number of jobs that can be stored in the cache (Refer Table 5.2). The difference in byte hit rates of GRV and LRU-Bundle is less than 5% for all cache sizes. Though GRV involves prefetching of data and LRU-Bundle does not, they seems to provide similar byte hit rates. We also see that LRU-Bundle has higher byte hit rate than GRV for cache sizes of 25 TB and 50 TB by taking advantage of the temporal locality.

From figure 5.7, it is found that the byte hit rate of GRV is worse than all the other algorithms. This is because, the GRV algorithm clears more space than what is actually required by the job. The GRV algorithm looks at all the files that are not currently in

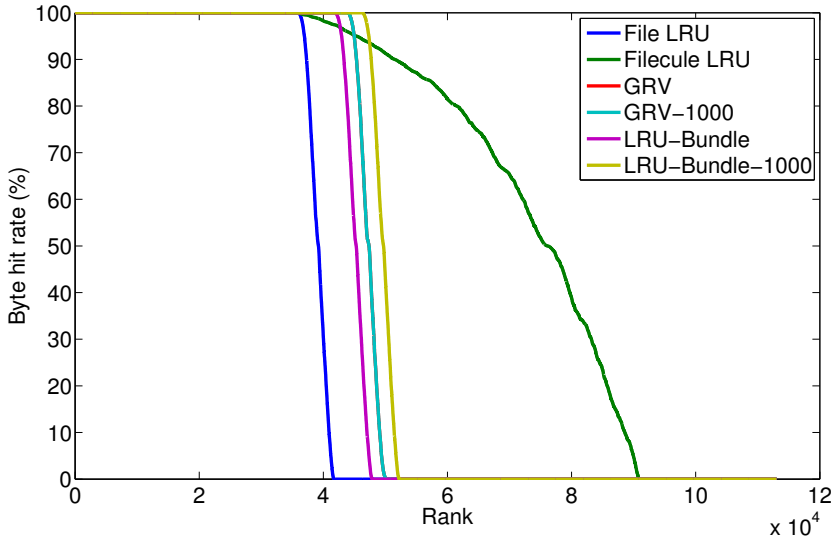

Figure 5.4 Byte Hit Rate for Cache Size of 5 TB. GRV = GRV-1000

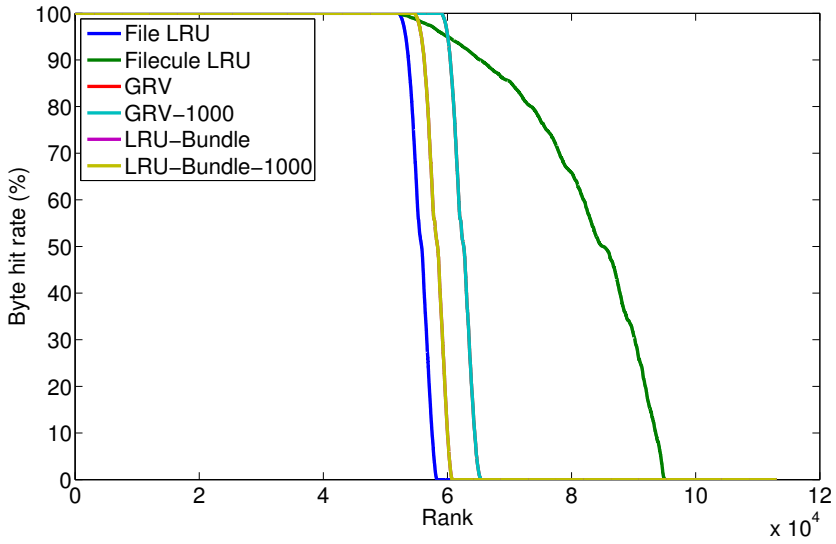

Figure 5.5 Byte Hit Rate for Cache Size of 10 TB. GRV = GRV-1000 and LRU-Bundle = LRU-Bundle-1000

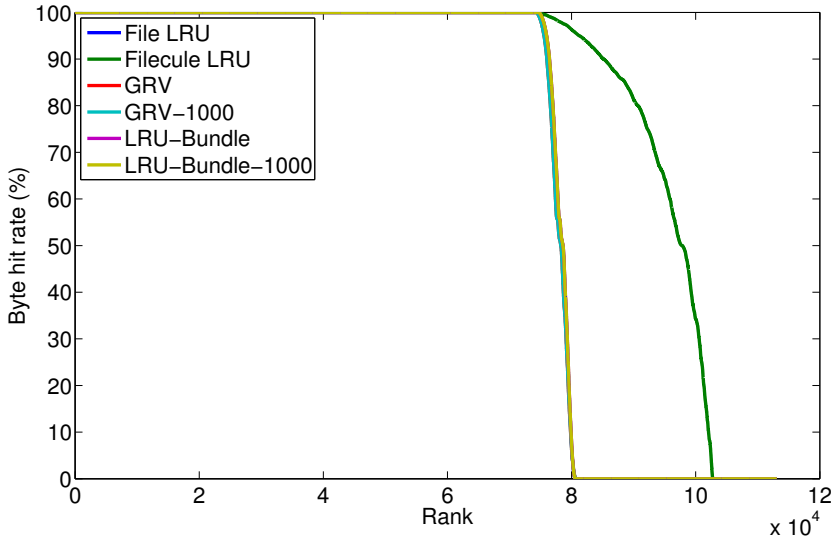

Figure 5.6 Byte Hit Rate for Cache Size of 25 TB. GRV = GRV-1000 and LRU-Bundle = LRU-Bundle-1000

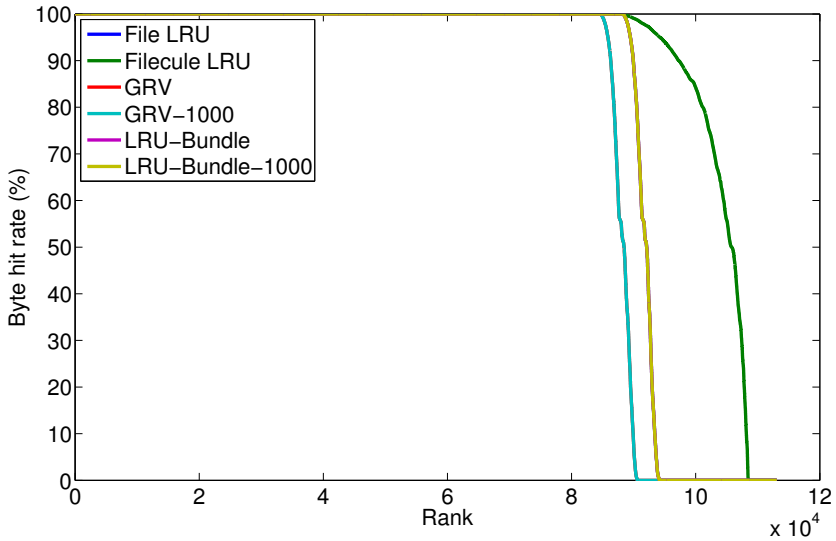

Figure 5.7 Byte Hit Rate for Cache Size of 50 TB. GRV = GRV-1000 and File LRU = LRU-Bundle = LRU-Bundle-1000

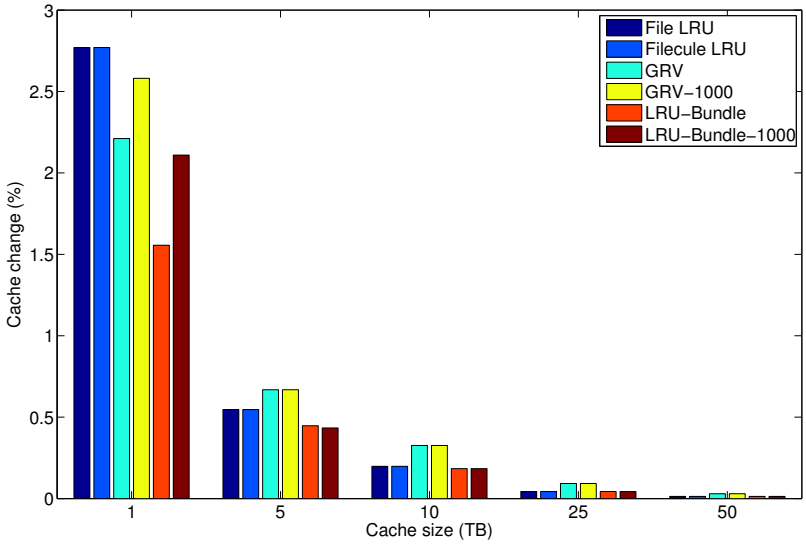

Figure 5.8 Average Percentage of Cache Change for Different Cache Sizes

use by any job, and determines whether or not to retain the file. This decision is based on the relative value of the request (Equation 5.2) that added this file to the cache. Thus the eviction of a file from the cache is not dependent on recency of use. The effect of eliminating more files than required and file elimination without taking into account recency of access is the cause for small byte hit rate.

5.5.2 Percentage of Cache Change

File LRU and Filecule LRU have the same amount of cache changes at all cache sizes because we use optimal filecules. There is no incorrect prediction, i.e., all the data that is prefetched is utilized by the job.

Figure 5.8 shows the average percentage of cache change for increasing cache sizes. Average percentage of cache change is best for the LRU-Bundle algorithm for all cache sizes. GRV has large cache changes for all caches sizes except at 1 TB, where File LRU and Filecule LRU have higher cache changes.

Queue freezing increases the percentage of cache change at 1 TB (Figure 5.9) because there is no change to the job scheduling order once the queue is frozen. Any job that arrives when the queue is frozen cannot take advantage of the contents of the cache.

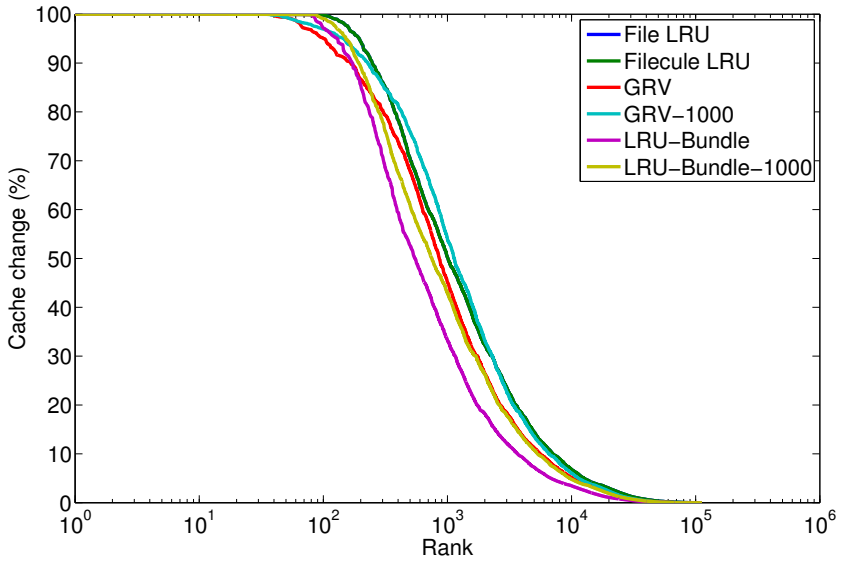

Figure 5.9 Percentage of Cache Change for Cache Size of 1 TB. File LRU = Filecule LRU

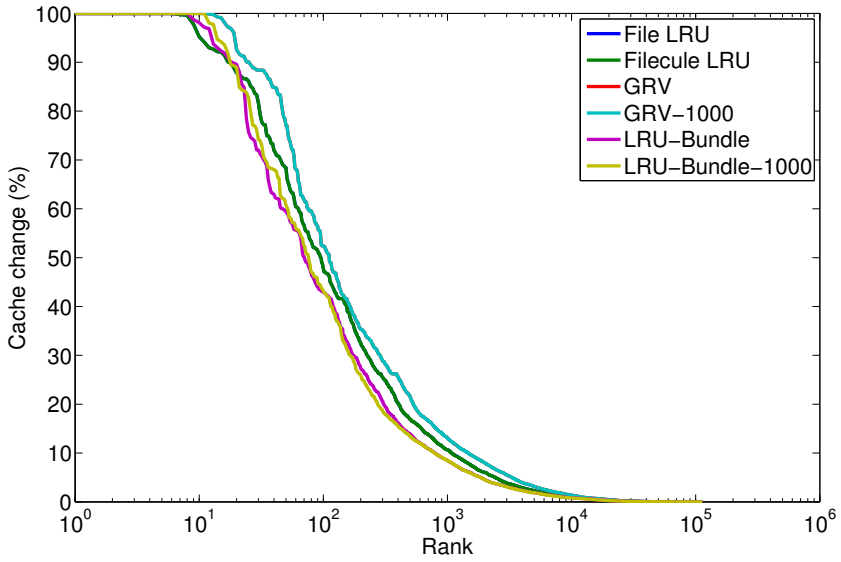

Figure 5.10 Percentage of Cache Change for Cache Size of 5 TB. File LRU = Filecule LRU and GRV = GRV-1000

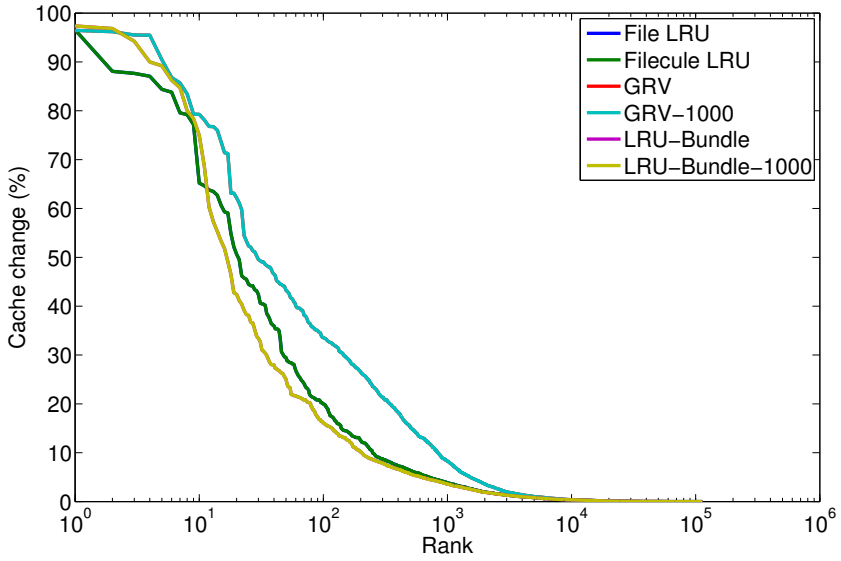

Figure 5.11 Percentage of Cache Change for Cache Size of 10 TB. File LRU = Filecule LRU, GRV = GRV-1000 and LRU-Bundle = LRU-Bundle-1000

At 10 TB (Figure 5.11), there are some changes to the cache made by LRU-Bundle algorithm that are higher than some of those made by File LRU and Filecule LRU. This can be an effect of job thrashing. Some jobs that need a lot of changes to the cache will be delayed by the LRU-Bundle algorithm. If they were scheduled with FCFS scheduling, they could have utilized the some of the contents of the cache. Since they were rescheduled to run at a later time, the amount of cache change needed at that time could be higher.

Figures 5.11, 5.12 and 5.13 show that the difference between the largest cache changes for GRV and the largest cache changes for LRU-Bundle increases with increasing cache size. As the cache size increases, the number of files that are not used by any job but are still retained in the cache increases. GRV algorithm is capable of replacing these huge available spaces though it is not required.

5.5.3 Job Waiting Time and Queue Length

When using FCFS scheduling algorithm, jobs can be delayed only due to lack of free space in the cache to load files for the job. In case of GRV algorithm, jobs can be delayed either due to lack of free cache space or due to reordering of jobs in the queue. The job

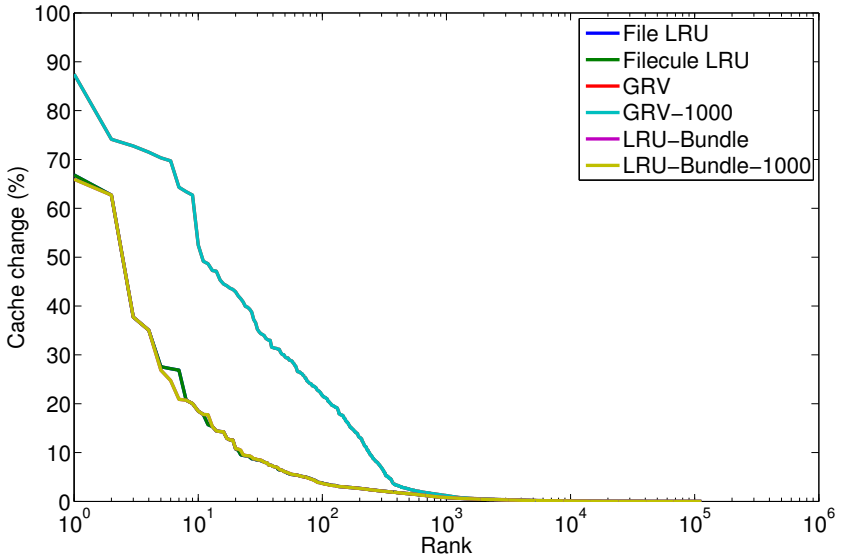

Figure 5.12 Percentage of Cache Change for Cache Size of 25 TB. File LRU = Filecule LRU, GRV = GRV-1000 and LRU-Bundle = LRU-Bundle-1000

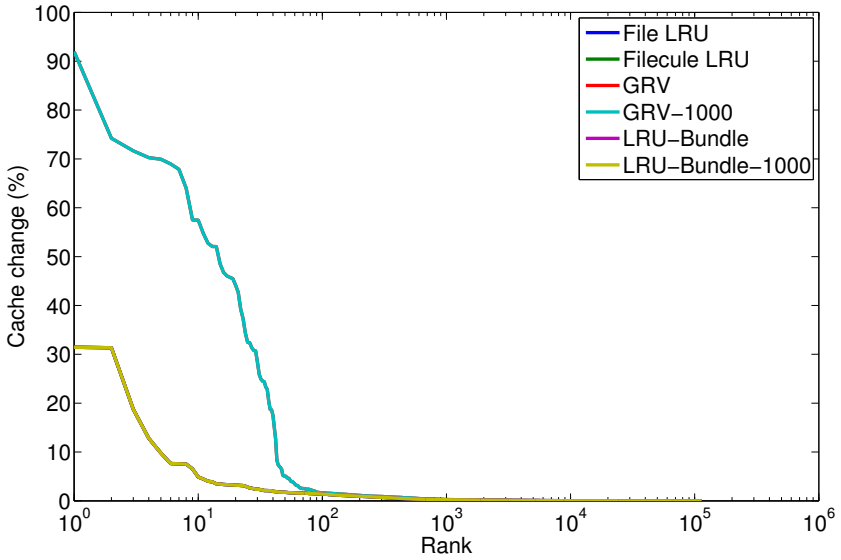

Figure 5.13 Percentage of Cache Change for Cache Size of 50 TB. File LRU = Filecule LRU = LRU-Bundle = LRU-Bundle-1000 and GRV = GRV-1000

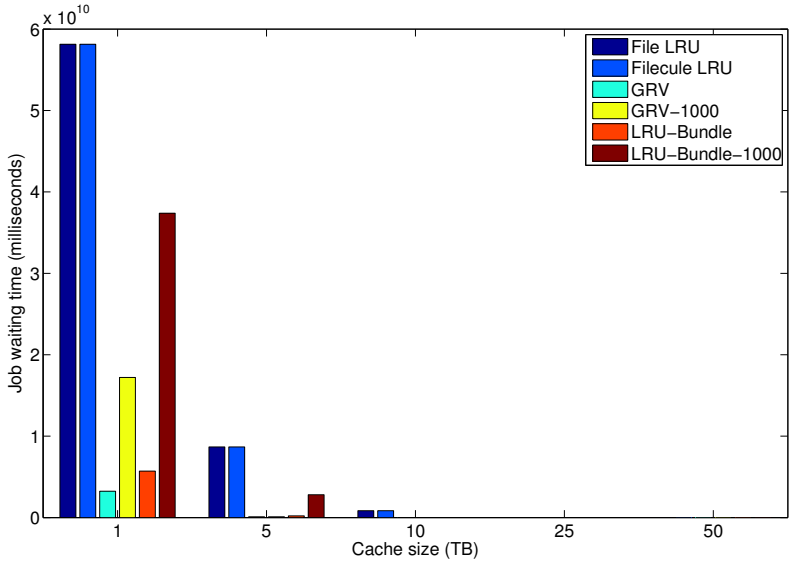

Figure 5.14 Average Job Waiting Time for Different Cache Sizes

waiting time for File LRU and Filecule LRU are equal because the jobs are processed in the same order (FCFS).

Figure 5.14 shows the average job waiting time for the various algorithms for increasing cache sizes. The job waiting time for File LRU and Filecule LRU are equal because the jobs are processed in the same order (FCFS). File LRU and Filecule LRU have the worse average waiting times because a lot of jobs are made to wait because of delay in scheduling the job at the head of the queue (FCFS). The Figure also shows that queue freezing increases the average job waiting time. This is because once the queue is frozen, any new job entering the queue is made to wait until all the jobs in the frozen section of the queue are scheduled to run.

Figures 5.16, 5.17 and 5.18 show the effect of job thrashing. It can be seen that when using GRV job scheduling algorithm, few jobs suffer longer delays than those observed when using FCFS. This is because some jobs are delayed until all the other jobs in the queue are scheduled. But it also shows that there are many jobs that suffer longer waiting time when using FCFS. This is because many jobs are made to wait until the job that is ahead in the queue is getting delayed due to lack of space in cache.

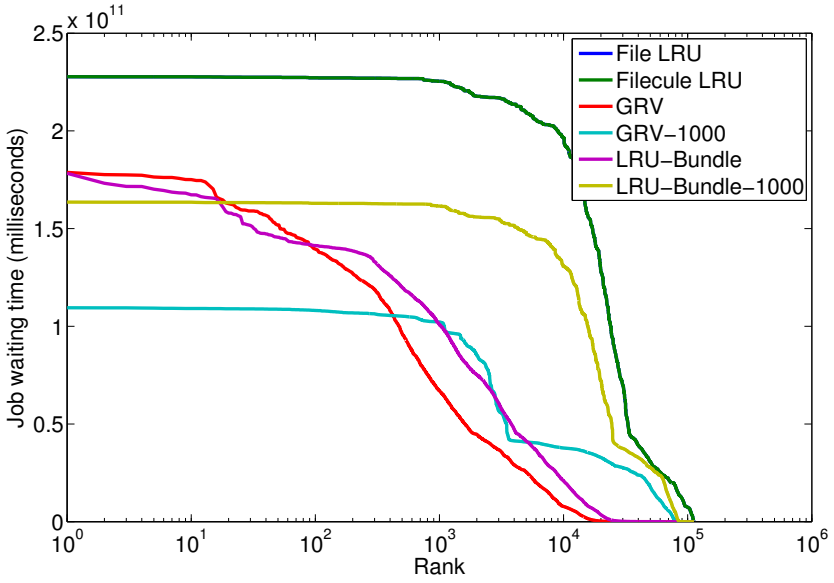

Figure 5.15 Job Waiting Time for Cache Size of 1 TB. File LRU = Filecule LRU

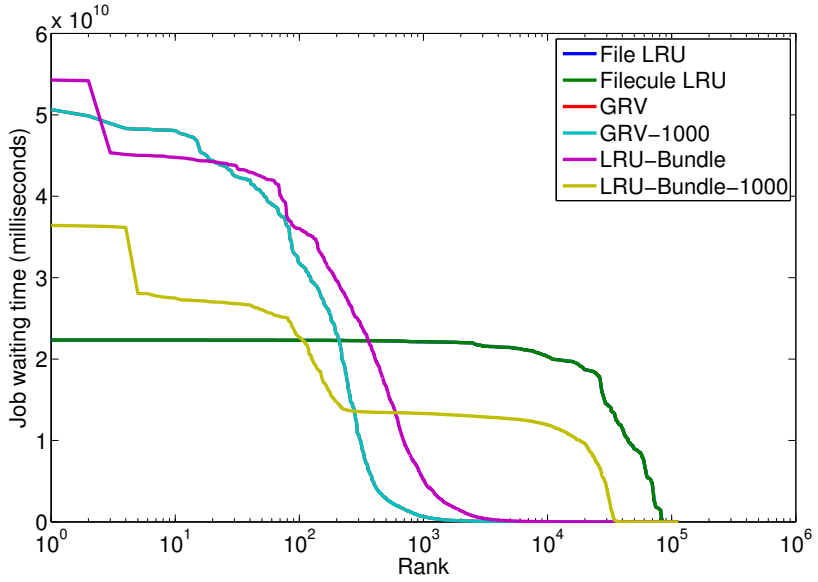

Figure 5.16 Job Waiting Time for Cache Size of 5 TB. File LRU = Filecule LRU and GRV = GRV-1000

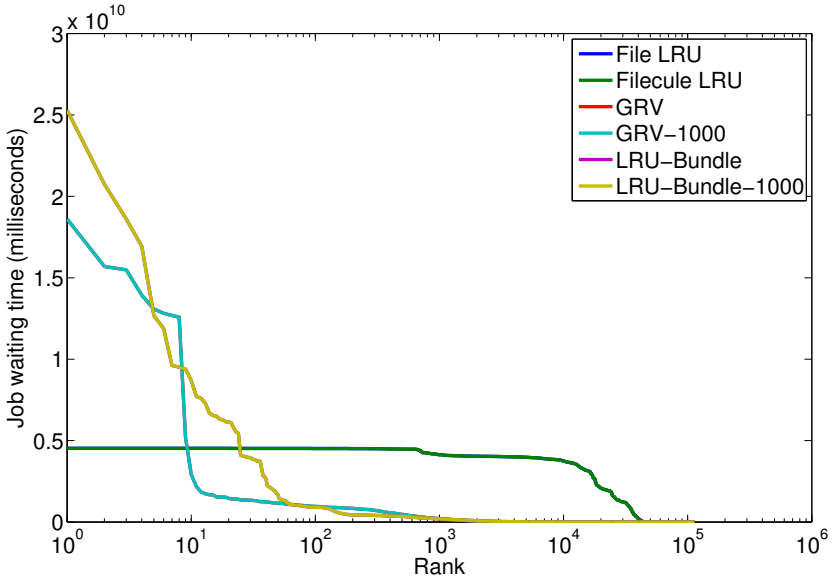

Figure 5.17 Job Waiting Time for Cache Size of 10 TB. File LRU = Filecule LRU, GRV = GRV-1000 and LRU-Bundle = LRU-Bundle-1000

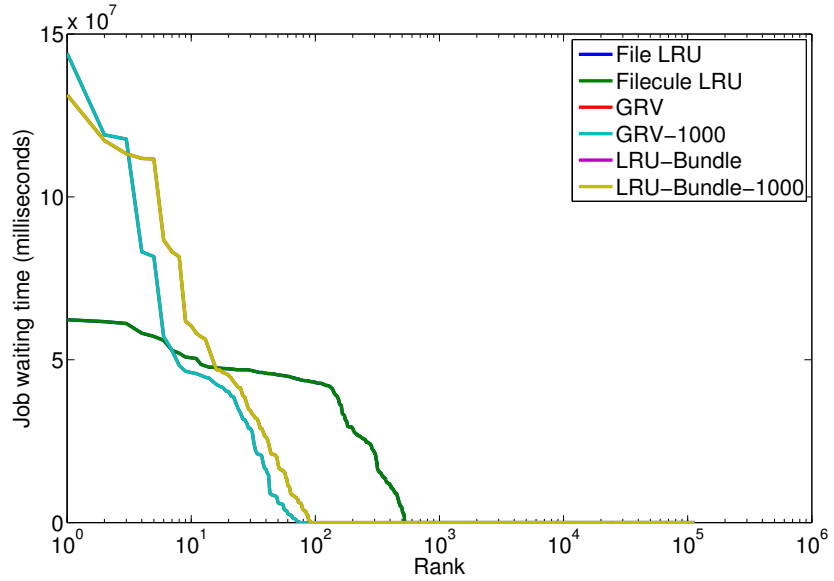

Figure 5.18 Job Waiting Time for Cache Size of 25 TB. File LRU = Filecule LRU, GRV = GRV-1000 and LRU-Bundle = LRU-Bundle-1000

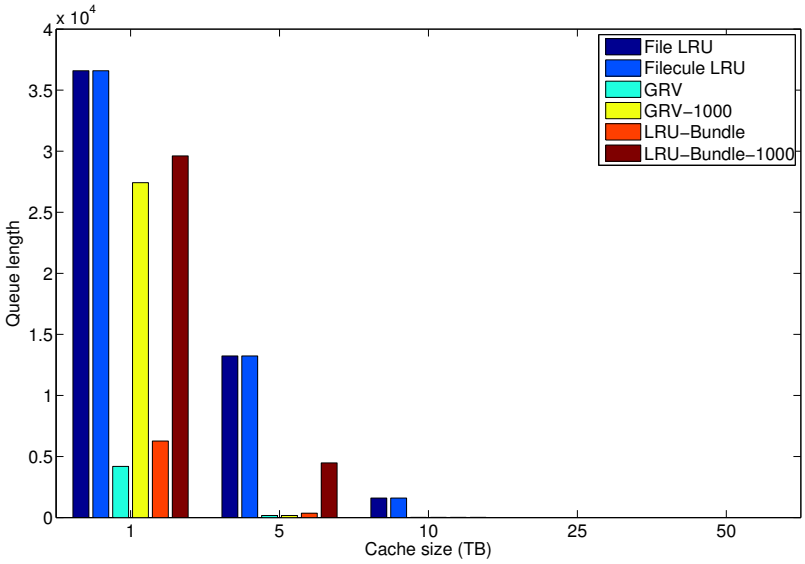

Figure 5.19 Average Queue Lengths for Different Cache Sizes

Figures 5.15 and 5.16 show that queue freezing avoids job thrashing, but also there is an increased number of jobs that have high job waiting times compared to those with no queue freezing.

The average queue length shown in Figure 5.19 is similar to the one observed in Figure 5.14. Figures 5.20 and 5.21 show that the queue gets longer with queue freezing. It is also interesting to see that at 10 TB, more jobs (68,415 jobs) are processed as soon as they enter the queue for File LRU and Filecule LRU than for GRV (61,771 jobs) or LRU-Bundle (50,289 jobs). But the average is affected by the long queues observed for the rest of the jobs for File LRU and Filecule LRU.

5.5.4 Scheduling Overhead

GRV and LRU-Bundle algorithms schedule jobs from the job waiting queue based on the the relative value of the request (Equation 5.2). Computing the relative value of the requests in the waiting queue is an overhead to the system. FCFS job scheduling does not incur any scheduling overhead. Hence the scheduling overhead for File LRU and Filecule LRU are 0.

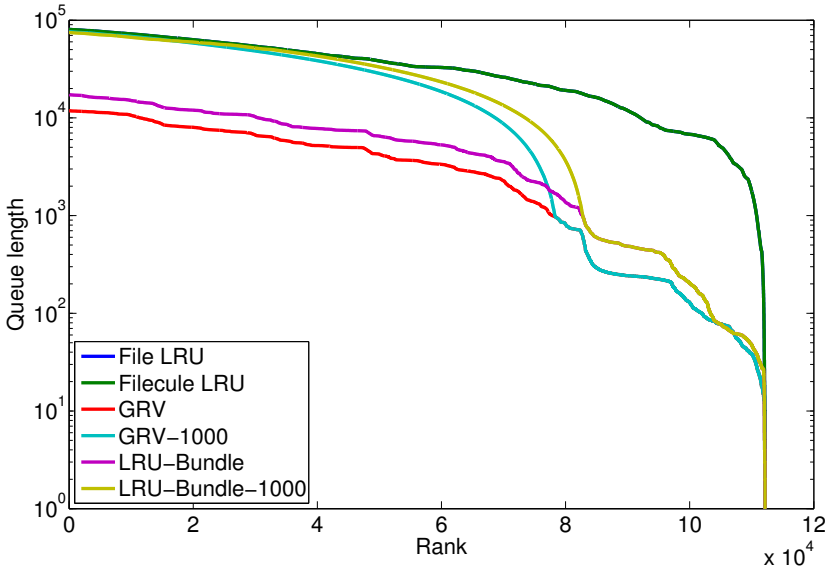

Figure 5.20 Queue Length for Cache Size of 1 TB. File LRU = Filecule LRU

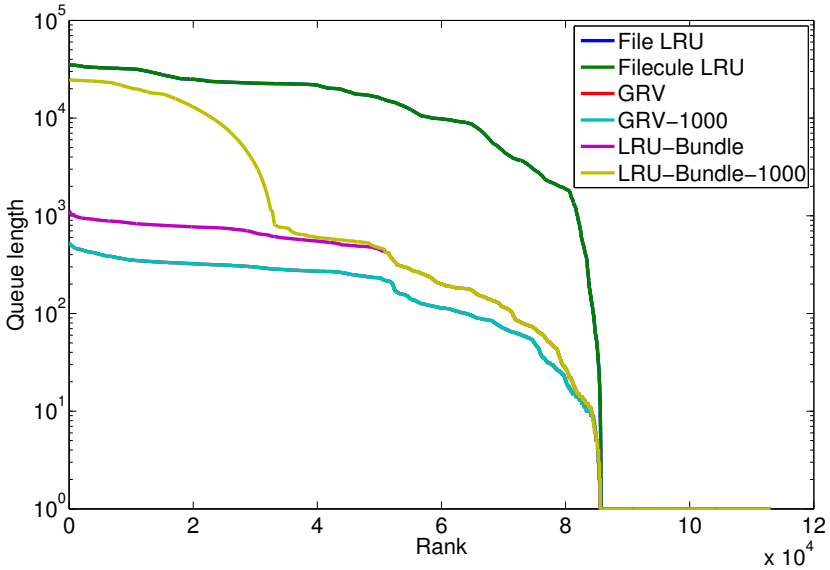

Figure 5.21 Queue Length for Cache Size of 5 TB. File LRU = Filecule LRU and GRV = GRV-1000

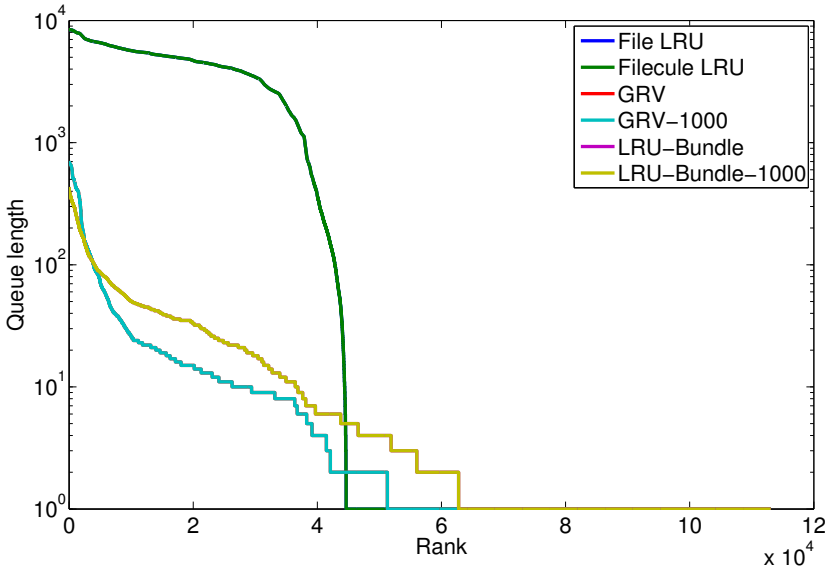

Figure 5.22 Queue Length for Cache Size of 10 TB. File LRU = Filecule LRU, GRV = GRV-1000 and LRU-Bundle = LRU-Bundle-1000

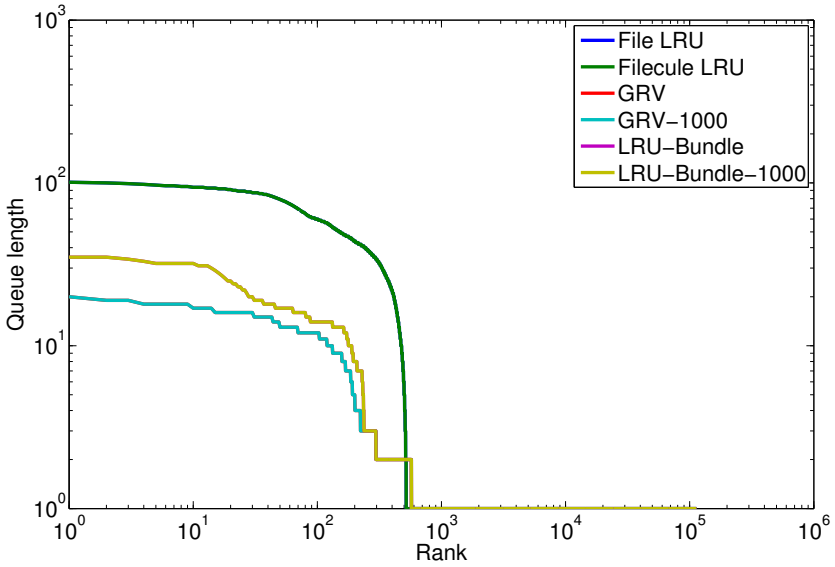

Figure 5.23 Queue Length for Cache Size of 25 TB. File LRU = Filecule LRU, GRV = GRV-1000 and LRU-Bundle = LRU-Bundle-1000

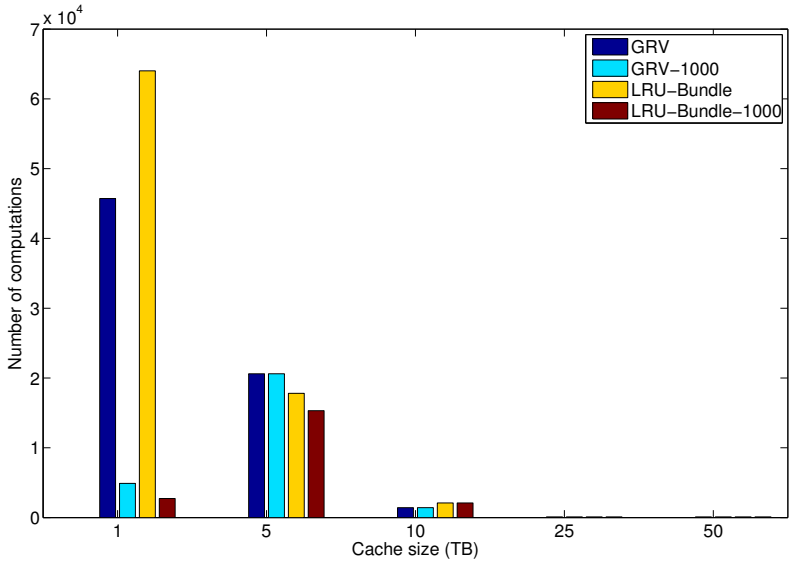

Figure 5.24 Average Scheduling Overhead for Different Cache Sizes

Figure 5.24 shows the average number of computations performed to schedule a job. It shows that different algorithms perform scheduling with small computational overhead for different cache sizes. For example, LRU-Bundle has more average computations than GRV for 1 TB and 10 TB cache sizes. The reverse is true for 5 TB cache size.

Figure 5.24 also shows that queue freezing reduces the computational overhead, because once the queue is frozen no more computation is done until all the jobs in the frozen part of the queue are scheduled to run. Figure 5.25 also shows the same effect. We expected the largest computational overhead for algorithms with queue freezing will be greater than the largest value without queue freezing because once the queue is unfrozen all the requests in the remaining queue is updated with their relative request values. Though a lot of computations are done as soon as the queue is unfrozen, it is not as large as the largest overhead for algorithms with no queue freezing.

Figure 5.28 shows that the overhead for GRV and LRU-Bundle are almost equal inspite of the queue length of LRU-Bundle being longer than GRV (Figure 5.23. At 50 TB (Figure 5.29), the computational overhead is equal for all the algorithms because all the jobs are scheduled to run as soon as they are submitted.

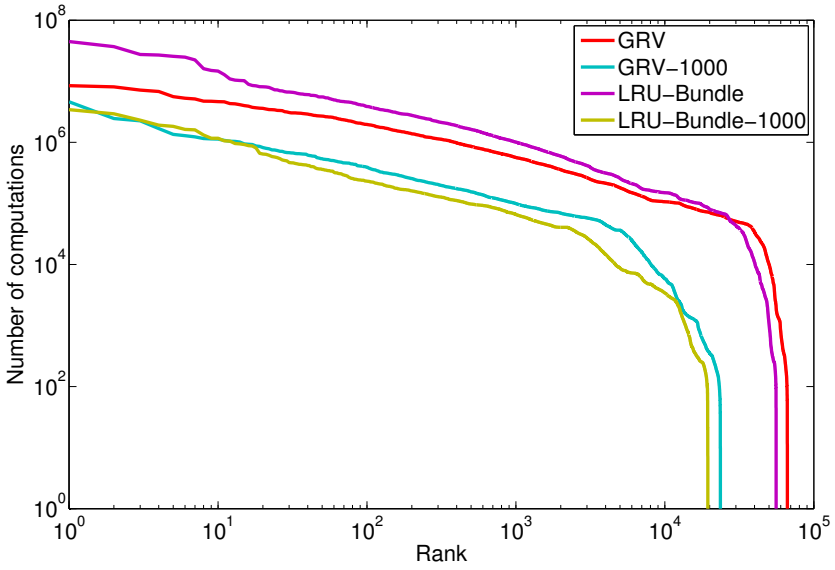

Figure 5.25 Scheduling Overhead for Cache Size of 1 TB. File LRU = Filecule LRU = 0

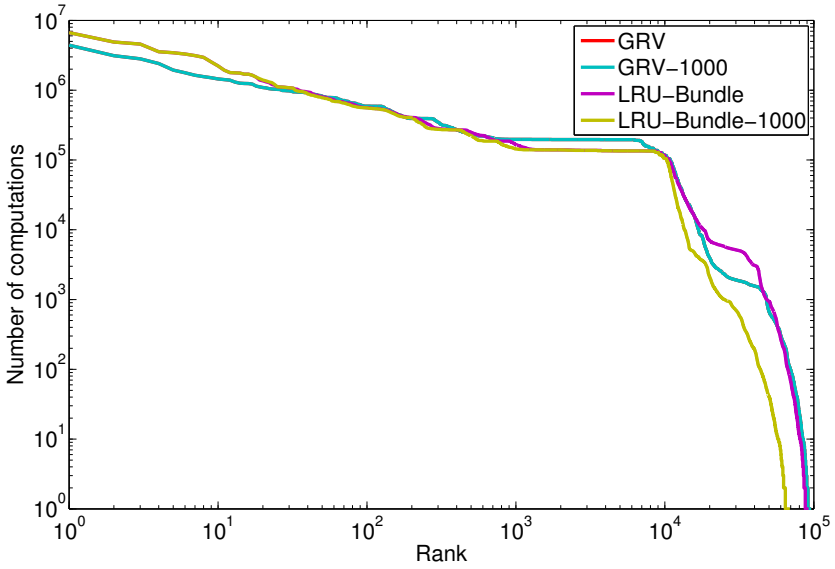

Figure 5.26 Scheduling Overhead for Cache Size of 5 TB. File LRU = Filecule LRU = 0 and GRV = GRV-1000

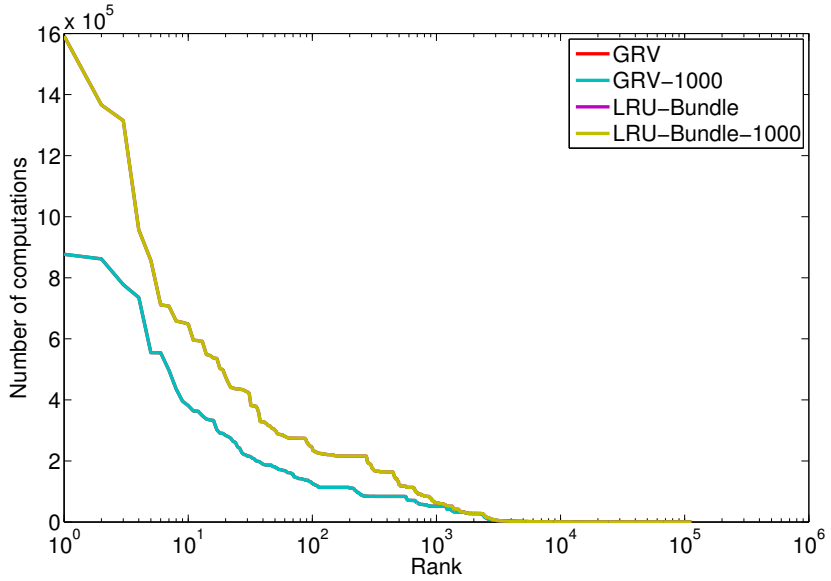

Figure 5.27 Scheduling Overhead for Cache Size of 10 TB. File LRU = Filecule LRU = 0, GRV = GRV-1000 and LRU-Bundle = LRU-Bundle-1000

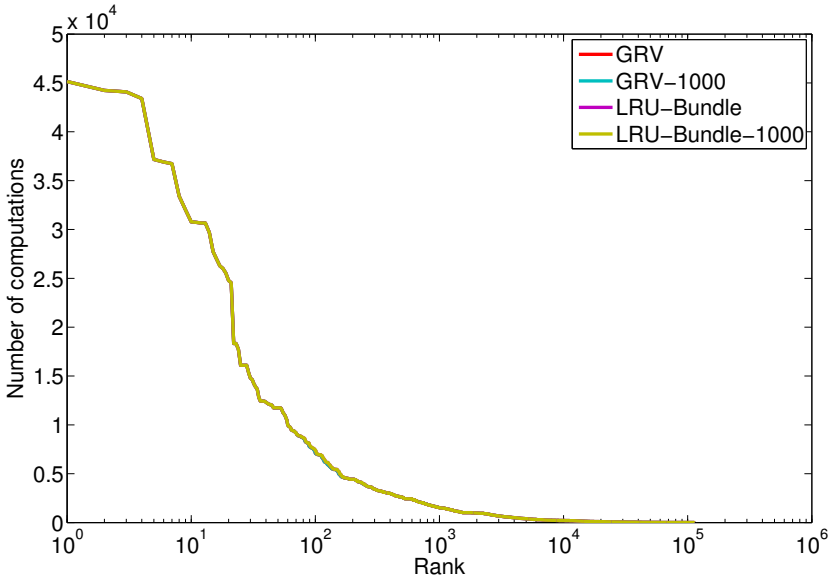

Figure 5.28 Scheduling Overhead for Cache Size of 25 TB. File LRU = Filecule LRU = 0, GRV = GRV-1000 and LRU-Bundle = LRU-Bundle-1000

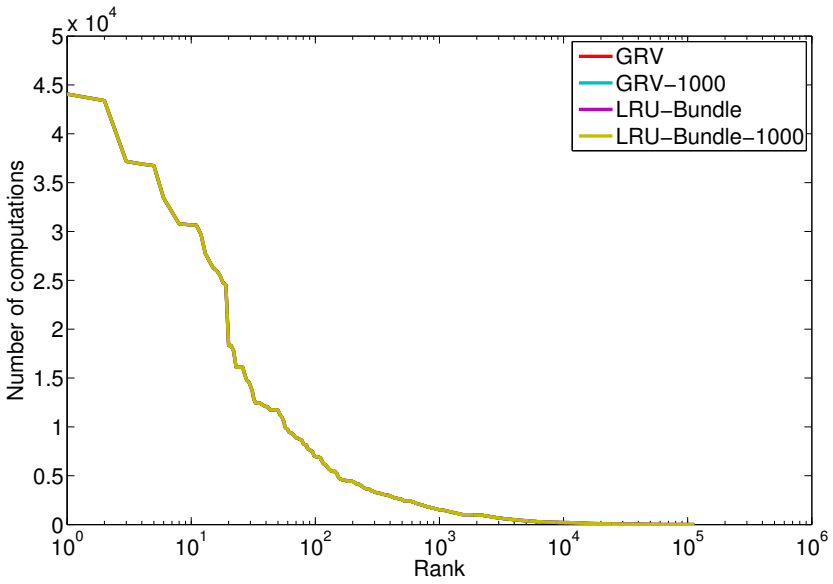

Figure 5.29 Scheduling Overhead for Cache Size of 50 TB. File LRU = Filecule LRU = 0 and GRV = GRV-1000 = LRU-Bundle = LRU-Bundle-1000

The computational overhead can be further reduced for all the above algorithms by not computing the relative value of an incoming job when there is no other job in the queue and when there is enough space to process the job immediately.

5.6 Summary

Table 5.3 lists the various metrics and the algorithm that performs best for that metric. It shows that LRU is a good cache replacement algorithm for the scientific workload used for these experiments. This is in agreement with the temporal locality of the workload. The FCFS job scheduling algorithm is not suitable because many jobs are delayed due to jobs at the head of the queue. LRU-Bundle performance measured with all the metrics listed in Section 5.3 takes advantage of the temporal locality in the workload and also provides short job waiting times. Predicting data usage using filecule definitions provides best byte hit rates.

The average scheduling overhead of LRU-Bundle is smaller than that of GRV. Apart from the computational overhead, GRV also has another overhead to calculate the optimal set of files that needs to be prefetched into the cache. The number of computations to

Table 5.3 Summary of Results on Caching and Scheduling Algorithms

Metric	Algorithm with the best performance
Byte hit rate	Filecule LRU
Percentage of cache change	LRU-Bundle
Job Waiting Time	GRV
Queue Length	GRV
Scheduling Overhead	File LRU and Filecule LRU

identify the optimal set of files to be loaded into the cache is dependent on the amount of information stored about jobs in the history. This overhead does not exist for LRU-Bundle.

The experimental results presented in this chapter suggest that by combining LRU-Bundle cache replacement algorithm with prefetching based on filecules may provide even better performance than the algorithms mentioned in Section 5.2.

CHAPTER 6

IMPACT OF HISTORY WINDOW ON FILECULE IDENTIFICATION

Data prefetching [10, 46, 48, 42, 44, 43] methods use recency and frequency of data usage from past requests to predict what data will be requested in the future. The important parameter that decides the performance of the prefetch is dependent on the amount of requests used from history. The requests from a period that is far back in the past should have low impact on the predictions and the requests from near past should have high impact. Maintaining all the requests from the past is practically impossible due to storage restrictions. The long processing time of long history of requests may add significant overhead to the system.

This chapter discusses the effect of the history window that is used to identify filecules. Optimal filecules were formed using a file request information from January 2003 to March 2005. This is the grouping that should be used during this 27 month period to achieve the best byte hit rates (Results using optimal filecules is shown in Section 5.5.1). An appropriate history window that predicts filecules closer to the optimal filecules needs to be identified.

The relationships between files can change over a period of time. Two files that have high correlation (always requested together by any job) during a period of time need not maintain their correlation during a later period in time. The reverse can also be true where two unrelated files might have stronger relationships in the future. A good history window should be capable of grouping files that are popular in the present and also be able to identify this transitioning relationships between files.

We experiment by using 1-month window. We compare the filecules identified after 1 month with the optimal set of filecules. Table 6.1 shows how many filecules identified match with the optimal filecules. The mismatch is due to the filecules identified during

Table 6.1 Comparison of 1-month Filecules and Optimal Filecules. Filecules formed with data used during January 2003 are compared with the optimal filecules identified with data used from January 2003 to March 2005

Category	# of filecules	% of filecules	% of bytes
Match	2,111	71.1	25
Mismatch	858	28.9	75

Table 6.2 Comparison of Filecules Identified in 2 Consecutive 1-month Windows. Filecules formed with data used during January 2003 is compared with the filecules formed with data used during February 2003

Category	# of filecules	% of filecules	% of bytes
Matching	2,162	72.8	13.67
Mismatch	583	19.6	60.68
Not represented	224	7.5	25.65

1-month window being larger than the optimal filecules. The percentage of bytes that do not form optimal filecules is high. The information available during one month is not enough to identify optimal filecules. Some of the optimal filecules are not represented in the 1-month window because the files in those optimal filecules were not requested during this 1-month window (not included in Table 6.1).

Table 6.2 compares the filecules formed during two consecutive 1-month windows. The percentage of bytes that do not match is high (60.68%). This shows that the relationships between files have changed within one month. It also shows that a considerable amount of bytes (25.65%) are not requested during the second month.

Data from the Tables 6.1 and 6.2 clearly illustrates that the window used to identify filecules should identify the transitioning relationships (decrease the mismatch between two consecutive windows) and the correct relationships (increase the match between one window and optimal filecules).

6.1 Filecule LRU Using 1-month Window

In order to simulate the effect of using 1-month window, we ran the cache simulation for the first month (January 2003). Before adding jobs for the next month (February 2003) to the waiting queue, we identify filecules using the history of jobs from January 2003.

Table 6.3 Comparison of Byte Hit Rate of Filecule LRU Using 1-month Window with File LRU and Filecule LRU Using Optimal Filecules

Category	File LRU	Filecule LRU with optimal filecules
% of jobs with equal byte hit rate	96.52	83.55
% of jobs with better byte hit rate	3.30	2.55
% of jobs with worse byte hit rate	0.17	14.89

Similar filecule identification is performed at the end of each month before scheduling a job from the next month. This Section will compare the byte hit rate and percentage of cache change per job obtained using File LRU, Filecule LRU using optimal filecules and Filecule LRU using 1-month window for filecule identification. The comparison will show that the performance of Filecule LRU using 1-month window is better than that of File LRU and worse than Filecule LRU using optimal filecules.

Figure 6.1 shows the difference in byte hit rate between Filecule LRU with 1-month window and File LRU. Most of the data points are found in the upper part of the graph, showing that byte hit rate per job for Filecule LRU with 1-month window is generally better than that of File LRU. This indicates that using a window of history to identify filecules improves the byte hit rate. Figure 6.2 shows the difference in byte hit rate between Filecule LRU with optimal filecules and Filecule LRU with 1-month window.

Filecule LRU with optimal filecules has better byte hit rate per job than Filecule LRU with 1-month window. Table 6.3 shows the percentage of jobs using Filecule LRU with 1-month window that have equal, worse and better byte hit rates compared to File LRU and Filecule LRU. There are certain sections of the plot in Figure 6.2 where the 1-month filecules have better hit rates than the optimal filecules. For those periods, the filecules using 1-month window define the best relationships. It indicates that there are some trade-offs when using long history to identify filecules, where we use information about transitioning relationships. Certain jobs can benefit from relationships identified during shorter windows.

The percentage of cache change per job when using File LRU and Filecule LRU algorithm with optimal filecules is the same. Figure 6.3 shows the difference between percentage of cache change between Filecule LRU using 1-month window and File LRU. The difference

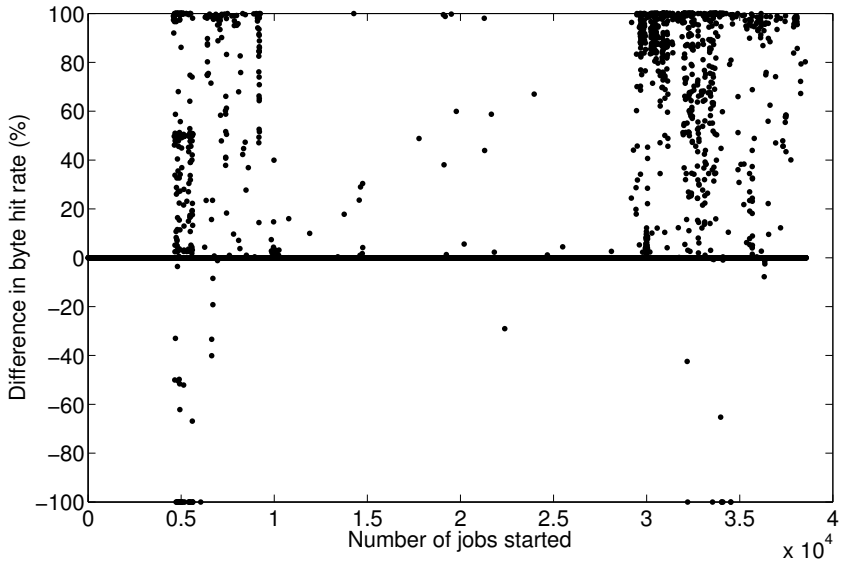

Figure 6.1 Difference in Byte Hit Rate Between Filecule LRU with 1-month Window and File LRU. File LRU has higher byte hit rates for 66 jobs. Filecule LRU with 1-month window has higher byte hit rates for 1,274 jobs. Equal byte hit rates observed for 37,225 jobs. Overall Filecule LRU with 1-month has better byte hit rates than File LRU

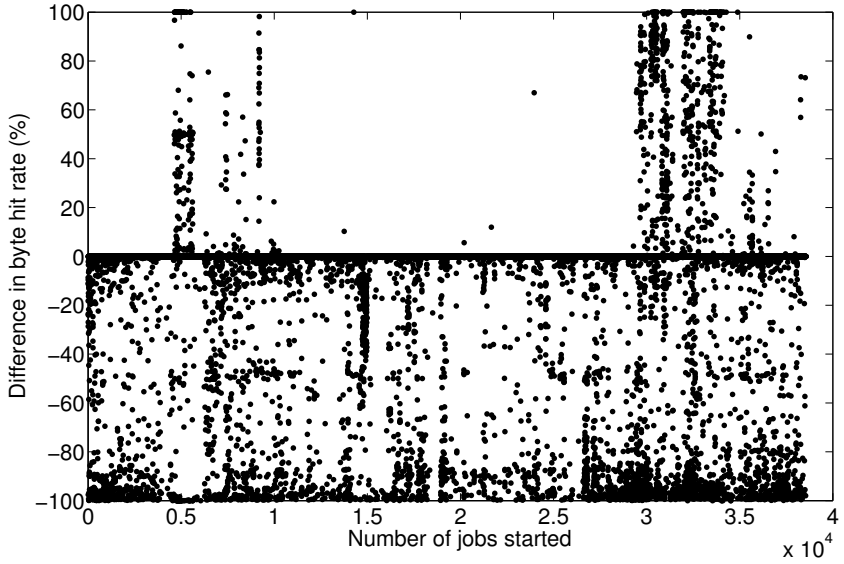

Figure 6.2 Difference in Byte Hit Rate Between Filecule LRU with Optimal Filecules and Filecule LRU with 1-month Window. Filecule LRU with optimal filecules has higher byte hit rates for 5,357 jobs. Filecule LRU with 1-month window has higher byte hit rates for 985 jobs. Equal byte hit rates observed for 32,223 jobs. Overall Filecule LRU with optimal filecules has better byte hit rates than Filecule LRU with 1-month window.

Table 6.4 Comparison of Percentage of Cache Change of Filecule LRU Using 1-month Window with File LRU and Filecule LRU Using Optimal Filecules. The percentage of cache change for File LRU is equal to the percentage of cache change for Filecule LRU with optimal filecules.

Category	File LRU
% of jobs with equal cache change	86.57
% of jobs with better cache change	8.04
% of jobs with worse cache change	6.47

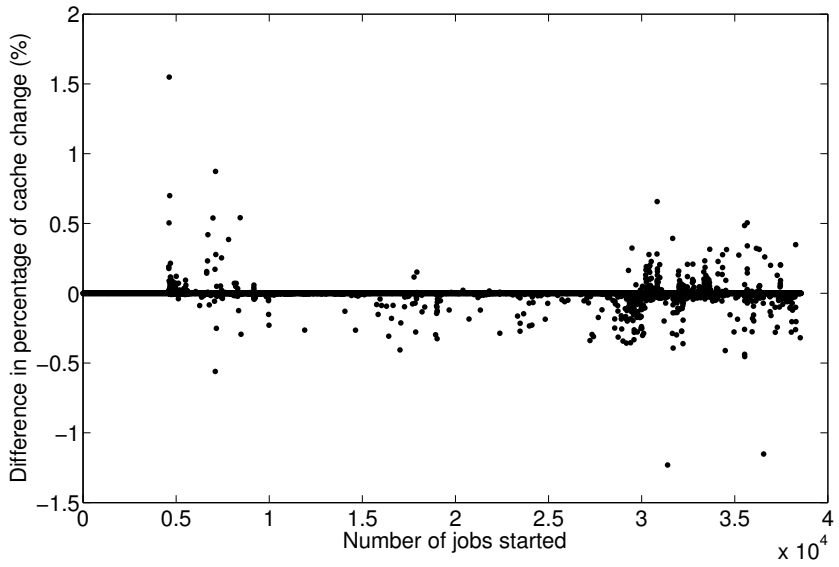

Figure 6.3 Difference in Percentage of Cache Change Between Filecule LRU with 1-month Window and File LRU. The percentage of cache change for File LRU is equal to the percentage of cache change for Filecule LRU with optimal filecules.

in percentage of cache change is not substantial. Table 6.4 shows the percentage of jobs using Filecule LRU with 1-month window that have equal, worse and better percentage of cache change compared to File LRU and Filecule LRU with optimal filecules. The number of jobs using Filecule LRU with 1-month window that cause larger changes to the cache (2,495 jobs) and those that cause smaller changes (2,684 jobs) compared to File LRU are almost equal. Figure 6.3 also shows that the maximum difference in percentage of cache change is small ($< 2\%$).

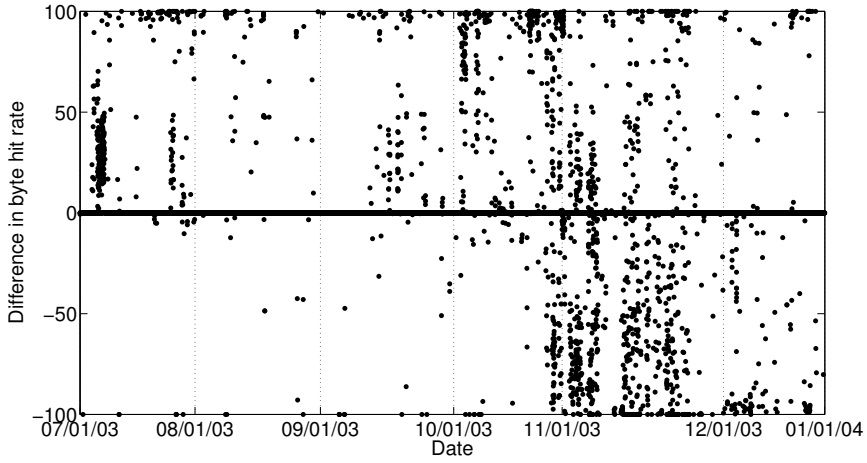

Figure 6.4 Difference in Byte Hit Rate Between Filecule LRU with 6-month Window and 1-month Window. 22,499 jobs have equal byte hit rates. Filecule LRU with 6-month window has lower byte hit rate for 1,005 jobs. Filecule LRU with 1-month window higher byte hit rate for 888 jobs.

These results (presented in Figures 6.1, 6.2 and 6.3) show that smaller window size for filecule identification in Filecule LRU still leads to better performance than File LRU. In the next set of experiments, we will see the impact of increasing window sizes.

6.2 Impact of Window Size in Filecule LRU

Small history windows form fewer filecules but with more files. This is due to the small number of jobs that influence the identification of filecules. As the number of jobs used for identifying filecules increases, the filecule definitions are more accurate. When only a small number of jobs are used for filecule identification, it will provide limited information about correlation between filecules. Hence, files that are not correlated may end up being part of a filecule. When this occurs, the efficiency of the prediction decreases. When using disjoint windows for identifying filecules, filecule information is lost from one window to the next. We ran experiments with different window lengths: 1 month and 6 month windows. We compared the difference in byte hit rates and percentage of cache change for each of these windows.

Table 6.5 Comparison of Byte Hit Rate of Filecule LRU Using 6-month Window and Filecule LRU Using 1-month Window

Category	Filecule LRU with 1-month Window
% of jobs with equal cache change	92.24
% of jobs with better cache change	4.12
% of jobs with worse cache change	3.95

Figure 6.4 shows the difference between byte hit rate obtained using Filecule LRU using 6-month window and Filecule LRU using 1-month window. Most of the jobs have the same byte hit rates for both windows. This shows that some filecules identified using 1-month window are similar to those identified using 6-month window. Table 6.5 shows the percentage of jobs that have equal, worse and better byte hit rates between Filecule LRU with 6-month window and Filecule LRU with 1-month window. Filecule LRU with 6-month window has better byte hit rates than those with 1-month window up to November 2003. During the month of November 2003, the filecules identified with 1-month window predict data usage better than the filecules identified with 6-month window. November 2003 is the reason for almost equal percentages of jobs with better cache change and jobs with worse cache change (Table 6.5). During all the regions except November 2003 and December 2003, Filecule LRU with 6-month window has higher byte hit rate than that of 1-month window. When using 1-month window, a lot of information about filecules formed during the first 5 months is lost. There is no prediction about those files. But there is also a significant amount of jobs that have smaller byte hit rates with 6-month window than byte hit rates with 1-month window. There are almost twice the number of jobs that arrived during that 1-month window (November 2003) which shows negative difference. Table 6.6 shows the number of jobs per month for the 6 months we have compared. This result suggests that we need to study if the window used to identify filecules should be based on the number of jobs processed since last filecule identification. For example, identify filecules every 1,000 jobs or so.

Figure 6.5 shows the difference in percentage of cache change per job between Filecule LRU with 6-month window and 1-month window. It also shows different behavior during November 2003 compared to the rest of the months. Throughout the entire period of

Table 6.6 Number of Jobs Per Month

Month	Number of Jobs
Jul 2003	3,762
Aug 2003	4,109
Sep 2003	4,365
Oct 2003	3,552
Nov 2003	5,286
Dec 2003	3,318

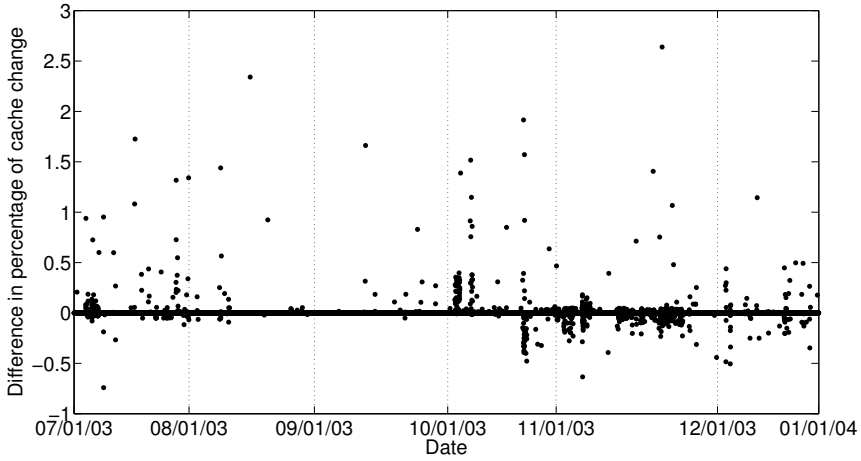

Figure 6.5 Difference in Percentage of Cache Change Between Filecule LRU with 6-month Window and 1-month Window

6 months of comparison, there is no significant difference in percentage of cache change between the two different windows: the largest difference being 2.6%. This need not be an effect of the algorithm itself. It can be because of the difference in the order in which the system chooses to add files to cache which affects the recency of access and this in turn affects the order in which files are evicted from the cache. This shows that unwanted files are not prefetched irrespective of the window length.

6.3 Summary

Based on the Figures 6.4 and 6.5, it is found that selecting an appropriate window to identify filecules, a length of history that will group files that are popular in the present and

also identify transitioning relationships between files, is important for cache performance (measured in byte hit rate per job).

Another direction suggested by our study is that performance is influenced by time locality, and thus sliding windows may lead to more adaptive grouping of files into filecules. Another observation is that instead of fixing the window size to a time interval, a window size dictated by job-interarrival time could improve caching performance. Another parameter that may dictate window sizes may be determined the transition rate of file popularity [29].

CHAPTER 7

CONCLUSION

We analyzed traces from a large high-energy physics collaboration focusing on aspects related to data usage. We propose a new abstraction for data management, namely filecules, and show that it is more efficient than traditional one-file data granularity for data caching.

Using stack depth analysis, we showed that this scientific workload exhibits good temporal locality (Section 5.1). Hence, algorithms that use recency of data usage needs to be used in such systems.

We proposed and evaluated a new combination of caching with job scheduling, LRU-Bundle (Chapter 5). We compared the performance of LRU-Bundle with LRU and GRV cache replacement algorithms by simulating disk cache events using real traces from the DZero Experiment. Our experiments show that LRU-Bundle provides better byte hit rates compared to File LRU (4%-106%) and GRV (4%-8%), and significantly shorter (3%-103% compared to File LRU) job waiting times. LRU-Bundle algorithm transfers less data (30% to 56%) from remote storage to SRM disk cache compared to GRV. In CMS grid [20], EU data grid [16] and Grid PP [21], the bandwidth between nodes varies from 10 Mb/s to 10 Gb/s. Most of the links being less than or equal to 1 Gb/sec. 30% of 1 TB amounts to 307 GB. The time taken to transfer 307 GB over a 10 Mb/s link is 29 days and 40 mins over 1 Gb/s link. When LRU-Bundle algorithm is used, the data transfer time is reduced by 40 mins to 29 days.

We also studied the effect of history window in identifying filecules and their impact on caching. We observed that small window sizes identify filecules that are large and large window sizes creates filecules of smaller sizes. We show that choosing an appropriate window is essential for the cache performance (measured in byte hit rate per job). More analysis needs to be performed using overlapping windows and choosing window length

based on usage patterns (such as number of jobs submitted, percentage of cache misses etc.)

This research leads to a new set of questions left for future work. What is the effect of identifying filecules that overlap instead of using disjoint filecules? From the discussion in Section 4.2.2, we see that files with high popularity do not group into large filecules. This may be due to some popular files being used along with different sets of files. For example, a popular file F may be used along with two different sets of files namely $\{F_A, ..., F_Z\}$ and $\{F_a, ..., F_z\}$ in two different jobs. This will lead to identifying 3 filecules: one mono-file filecule with F, a filecule with set of files $\{F_A, ..., F_Z\}$ and another filecule with $\{F_a, ..., F_z\}$. If we allow overlap between filecules, then instead of the above 3 filecules, 2 filecules can be identified with one consisting of $\{F, F_A, ..., F_Z\}$ and another consisting of $\{F, F_a, ..., F_z\}$. In other words, files with correlation coefficients less than 1 can be grouped into filecules. This will be analogous to the file groups identified in [30] using cosine correlation.

Another direction for future work is to apply filecules for data replication and placement, as discussed in Chapter 4. In that case, we would need to consider the benefits of using filecules and the tradeoffs in replication costs.

Finally, we would like to verify the generality of the patterns we identified on other scientific workloads. Recent efforts led to the creation of a Grid Workload Archive [32] that may make available other relevant traces in the near future.

REFERENCES

[1] The Collider Detector at Fermi National Accelerator Laboratory, http://www-cdf.fnal.gov/physics/public/public.html.

[2] The DZero Experiment, http://www-d0.fnal.gov.

[3] Deutches Elektronen-Synchrotron, http://www.desy.de.

[4] Thomas Jefferson National Accelerator Facility, http://www.jlab.org.

[5] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A. Fox. Caching proxies: Limitations and potentials. Technical report, Blacksburg, VA, USA, 1995.

[6] S. Acharya, B. Smith, and P. Parnes. Characterizing user access to videos on the world wide web. In *Proceedings of Multimedia Computing and Networking*, 2000.

[7] Lada Adamic, Bernardo Huberman, Rajan Lukose, and Amit Puniyani. Search in power law networks. *Physical Review. E*, 64:46135–46143, 2001.

[8] Charu G. Aggarwal and Philip S. Yu. On disk caching of web objects in proxy servers. In *CIKM '97: Proceedings of the sixth international conference on Information and knowledge management*, pages 238–245, New York, NY, USA, 1997. ACM Press.

[9] Virgilio Almeida, Azer Bestavros, Mark Crovella, and Adriana deOliveira. Characterizing reference locality in the www. Technical report, Boston, MA, USA, 1996.

[10] Ahmed Amer, Darrell D. E. Long, and Randal C. Burns. Group-based management of distributed file caches. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 525, Washington, DC, USA, 2002. IEEE Computer Society.

[11] Martin Arlitt, Rich Friedrich, and Tai Jin. Workload characterization of a web proxy in a cable modem environment. *SIGMETRICS Perform. Eval. Rev.*, 27(2):25–36, 1999.

[12] Martin Arlitt and Tai Jin. A workload characterization study of the 1998 world cup web site. Technical report, 1999.

[13] Martin F. Arlitt, Rich Friedrich, and Tai Jin. Performance evaluation of web proxy cache replacement policies. In *TOOLS '98: Proceedings of the 10th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, pages 193–206, London, UK, 1998. Springer-Verlag.

[14] Martin F. Arlitt and Carey L. Williamson. Internet web servers: workload characterization and performance implications. *IEEE/ACM Trans. Netw.*, 5(5):631–645, 1997.

[15] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in web client access patterns: Characteristics and caching implications. Technical report, Boston, MA, USA, 1998.

[16] William H. Bell, David G. Cameron, and A. Paul Millar. Optorsim: A grid simulator for studying dynamic data replication strategies. 17-4:403–416.

[17] A. Bestavros. Demand-based document dissemination to reduce traffic and balance load in distributed information systems. In *SPDP '95: Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, page 338, Washington, DC, USA, 1995. IEEE Computer Society.

[18] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM (1)*, pages 126–134, 1999.

[19] R. Brun and F. Rademakers. In *An object oriented data analysis framework*, 1996.

[20] D. G. Cameron, R. Carvajal-Schiaffino, A. P. Millar, C. Nicholson, K. Stockinger, and F. Zini. Evaluating scheduling and replica optimisation strategies in optorsim. In *Fourth International Workshop on Grid Computing, 2003*, pages 52–59.

[21] D. G. Cameron, R. Carvajal-Schiaffino, A. P. Millar, C. Nicholson, K. Stockinger, and F. Zini. Uk grid simulation benchmarks with optorsim. In *UK e-Science All Hands Conference*, Nottingham, 2003.

[22] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, 1997.

[23] Edith Cohen, Amos Fiat, and Haim Kaplan. Associative search in peer to peer networks: Harnessing latent semantics. In *Infocom*, San Fancisco, CA, 2003.

[24] Carlos Cunha, Azer Bestavros, and Mark Crovella. Characteristics of www client-based traces. Technical report, Boston, MA, USA, 1995.

[25] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS International conference on Measurement and Modeling of Computer Systems*, pages 59–70, New York, NY, USA, 1999. ACM Press.

[26] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150:1–4, 2001.

[27] P. Fuhrmann. dCache: the commodity cache. In *Twelfth NASA Goddard and Twenty First IEEE Conference on Mass Storage Systems and Technologies*, 2004.

[28] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *USENIX Annual Technical Conference*, pages 1–17, 1997.

[29] Adam Shaked Gish, Yuval Shavitt, and Tomer Tankel. Geographical statistics and characteristics of p2p query strings. In *IPTPS2007 - Proceedings of the 6th International Workshop on Peer-to-Peer Systems*, February 2007.

[30] Christos Gkantsidis, Thomas Karagiannis, Pablo Rodriguez, and Milan Vojnovic. Planet scale software updates. Technical report, 2006.

[31] Jim Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *USENIX Summer*, pages 197–207, 1994.

[32] The Grid Workloads Archive, http://gwa.ewi.tudelft.nl/.

[33] A. Iamnitchi and M. Ripeanu. Myth and reality: Usage behavior in a large data-intensive physics project, 2003.

[34] Adriana Iamnitchi and Ian Foster. Interest-aware information dissemination in small-world communities. In *High Performance Distributed Computing*, 2005.

[35] Adriana Iamnitchi, Matei Ripeanu, and Ian Foster. Small-world file-sharing communities. In *Infocom*, Hong Kong, China, 2004.

[36] Akamai Technologies Inc. Web application accelerator. Technical report, 2005.

[37] G. Irlam. Unix file size survey - 1993, 1993.

[38] L. Loebel-Carpenter, L. Lueking, C. Moore, R. Pordes, J. Trumbo, S. Veseli, I. Terekhov, M. Vranicar, S. White, and V. White. Sam and the particle physics data grid. In *In Computing in High-Energy and Nuclear Physics, Beijing, China*, 2001.

[39] Sape J. Mullender and Andrew S. Tanenbaum. Immediate files. *Softw. Pract. Exper.*, 14(4):365–368, 1984.

[40] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 297–306, New York, NY, USA, 1993. ACM Press.

[41] Ekow Otoo, Frank Olken, and Arie Shoshani. Disk cache replacement algorithm for storage resource managers in data grids. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–15, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[42] Ekow Otoo, Doron Rotem, and Alexandru Romosan. Optimal file-bundle caching algorithms for data-grids. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 6, Washington, DC, USA, 2004. IEEE Computer Society.

[43] Ekow Otoo, Doron Rotem, Alexandru Romosan, and Sridhar Seshadri. File caching in data intensive scientific applications. In *Data Management in Grids - Lecture Notes in Computer Science*, volume Volume 3836/2006, pages 85–99. Springer Berlin / Heidelberg, 2006.

[44] Ekow Otoo, Doron Rotem, and Sridhar Seshadri. Efficient algorithms for multi-file caching. In *Database and Expert Systems Applications - Lecture Notes in Computer Science*, pages 707–719. Springer Berlin / Heidelberg, 2004.

[45] Ekow Otoo and Arie Shoshani. Accurate modeling of cache replacement policies in a data grid. In *MSS '03: Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, page 10, Washington, DC, USA, 2003. IEEE Computer Society.

[46] Jim Pitkow and Mimi Recker. A simple yet robust caching algorithm based on dynamic access patterns. Technical Report GVU Technical Report, GIT-GVU-94-39, 1994.

[47] Arcot Rajasekar, Michael Wan, and Reagan Moore. Mysrb & srb: Components of a data grid. In *HPDC '02: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*, page 301, Washington, DC, USA, 2002. IEEE Computer Society.

[48] Margaret M. Recker and James E. Pitkow. Predicting document access in large multimedia repositories. *ACM Trans. Comput.-Hum. Interact.*, 3(4):352–375, 1996.

[49] Matei Ripeanu, Adriana Iamnitchi, and Ian Foster. Mapping the Gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *Internet Computing*, 6(1):50–57, 2002.

[50] Stefan Saroiu, Krishna P. Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy. An analysis of internet content delivery systems. *SIGOPS Oper. Syst. Rev.*, 36(SI):315–327, 2002.

[51] A. Shoshani, A. Sim, and J. Gu. Storage resource managers: Middleware components for grid storage, 2002.

[52] K. Sripanidkulchai. The popularity of gnutella queries and its implications on scalability, 2001.

[53] C. D. Tait and D. Duchamp. Detection and exploitation of file working sets. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS)*, pages 2–9, Washington, DC, 1991. IEEE Computer Society.

[54] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. File size distribution on unix systems: then and now. *SIGOPS Oper. Syst. Rev.*, 40(1):100–104, 2006.

[55] I. Terekhov. Meta-computing at d0. In *In Nuclear Instruments and Methods in Physics Research, Section A, NIMA14225*, volume 502/2-3, pages 402–406, 2002.

[56] Werner Vogels. File system usage in windows nt 4.0. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 93–109, New York, NY, USA, 1999. ACM Press.

[57] Neal E. Young. On-line file caching. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 82–86, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

APPENDICES

Appendix A Probability Distributions

A.1 Log Normal Distribution

A continuous distribution in which the logarithm of a variable has a normal distribution. The probability density function of a log normal distribution is given by

$$f(x;\mu,\sigma) = \frac{e^{\frac{-(\ln x-\mu)^2}{2\sigma^2}}}{x\sigma\sqrt{2\pi}} \tag{A.1}$$

for x¿0, where μ is the mean of the variable's logarithm and σ is the standard deviation of the variable's logarithm.

A.2 Log Logistic Distribution

A continuous distribution in which the logarithm of a variable has a logistic distribution. The probability density function of a logistic distribution is given by

$$f(x;\mu,s) = \frac{e^{\frac{-(x-\mu)}{s}}}{s\left(1+e^{\frac{-(x-\mu)}{s}}\right)^2} \tag{A.2}$$

where μ is the location parameter, s is the scale parameter and s¿0.

A.3 Generalized Pareto Distribution

The probability density function of generalized pareto distribution is

$$f(x;k,\sigma,\theta) = \left(\frac{1}{\sigma}\right)\left(1+k\frac{x-\theta}{\sigma}\right)^{-1-\frac{1}{k}} \tag{A.3}$$

where k is the shape parameter, $k\neq 0$, σ is the scale parameter and θ is the threshold parameter.

A.4 Hyper Exponential Distribution

The probability density function of a random variable X is

$$f_X(x) = \sum_{i=1}^{n} f_{Y_i}(y) p_i \tag{A.4}$$

where Y_i is an exponentially distributed random variable with rate parameter λ_i, and p_i is the probability that X will take on the form of the exponential distribution with rate λ_i.

A.5 Extreme Value Distribution

The probability density function of extreme value distribution is

$$f(x; \mu, \sigma) = \sigma^{-1} exp\left(\frac{x-\mu}{\sigma}\right) exp\left(-exp\left(\frac{x-\mu}{\sigma}\right)\right) \tag{A.5}$$

where μ is the location parameter and σ is the scale parameter.

A.6 Zipf Distribution

The zipf distribution follows power law. It is a discrete distribution with probability mass function

$$p(x; \alpha, n) = \frac{\frac{1}{x^{\alpha}}}{\sum_{i=1}^{n} \frac{1}{i^{\alpha}}} \tag{A.6}$$

where $x = 1, 2, ..., n$, $\alpha > 1$ and n is a positive integer.