Performance Isolation in Cloud Storage Systems

by

Akshay K. Singh

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Cloud computing enables data centres to provide resource sharing across multiple tenants. This sharing, however, usually comes at a cost in the form of reduced isolation between tenants, which can lead to inconsistent and unpredictable performance. This variability in performance becomes an impediment for clients whose services rely on consistent, responsive performance in cloud environments. The problem is exacerbated for applications that rely on cloud storage systems as performance in these systems is affected by disk access times, which often dominate overall request service times for these types of data services.

In this thesis we introduce MicroFuge, a new distributed caching and scheduling middleware that provides performance isolation for cloud storage systems. To provide performance isolation, MicroFuge's cache eviction policy is tenant and deadline-aware, which enables the provision of isolation to tenants and ensures that data for queries with more urgent deadlines, which are most likely to be affected by competing requests, are less likely to be evicted than data for other queries. MicroFuge also provides simplified, intelligent scheduling in addition to request admission control whose performance model of the underlying storage system will reject requests with deadlines that are unlikely to be satisfied.

The middleware approach of MicroFuge makes it unique among other systems which provide performance isolation in cloud storage systems. Rather than providing performance isolation for some particular cloud storage system, MicroFuge can be deployed on top of any already deployed storage system without modifying it. Keeping in mind the wide spectrum of cloud storage systems available today, such an approach make MicroFuge very adoptable.

In this thesis, we show that MicroFuge can provide significantly better performance isolation between tenants with different latency requirements than Memcached, and with admission control enabled, can ensure that more than certain percentage of requests meet their deadlines.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Prof. Bernard Wong and Prof. Khuzaima Daudjee for their patient guidance, insightful criticism and kind advice. I would also like to thank Prof. Paul Ward and Prof. Wojciech Golab for being my thesis readers.

Dedication

This thesis is dedicated to my parents who have supported me all the way since the beginning of my studies.

Also, this thesis is dedicated to my friends who has been a great source of motivation and encouragement.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cloud computing has had a transformative effect on how businesses host their online services and manage their computational needs. It reduces the start-up and operational cost of software services by commoditizing computing resources and enabling the clients to shrink or grow their capacity as needed. This gives more flexibility to organizations to experiment with new ideas or quickly deploy their products through the use of a variety of well-tested cloud-services provided on a pay-as-you-go basis. Cloud computing is especially useful for small and medium scale enterprises as it allows them to increase their peak-load capacity without incurring the prohibitive cost of building and maintaining their own infrastructure.

In a pay-as-you-go model, it is uncommon for all the clients to use their allocated resources at the same time. This allows cloud providers to statistically multiplex tenant services on to the same underlying infrastructure. By consolidating resources for large groups of tenants, a cloud provider can significantly reduce its hardware and operational costs. This in turn allows cloud providers to offer a price-competitive hosting service to their tenants. Unfortunately, such multi-tenant infrastructure often involves over-subscription of resources, which results in resource contention. Such contention results in poor and unpredictable performance if multiple tenants require the use of their resource reservations concurrently. Unpredictable performance may not affect some workloads (e.g., a map-reduce analytical workload), but it can be unacceptable for many important latency-sensitive workloads, such as web services and applications.

User studies have shown that web services' user requests are latency sensitive and should be satisfied in a specific amount of time, which is generally in few 100 ms [42]. The utility of servicing a request is significantly reduced after the deadline is expired, irrespective of how

close it is to the perfect response. Internet giants like Amazon and Google have realized that any degradation in page-load time is a major source of user dissatisfaction [36, 28]. With the Tolerable Wait Time (TWT) of an average web-user shrinking every year it is becoming increasingly difficult to meet these end-user response time SLOs, which is necessary for a web-service to remain competitive. Therefore, the need of a system which can help to meet these SLOs is important for such latency-sensitive workloads.

1.1 Resource Allocation in Multi-Tenant Environment

In multi-tenant environments, each tenant receives only a fraction of the total available resources and secures a fraction of their peak throughput. Performance in such a sharing model is affected by the workloads of the other concurrent clients, resulting in fluctuations in performance of clients. To have predictable performance in such environments, a workload should obtain sufficient resources for the expected performance, irrespective of the behaviour of other concurrent workloads in the system. While unpredictable performance in cloud environments is a well-known issue and is listed as one of the top obstacles in growth of cloud computing by Armbrust et al. [15], it is still not possible to provide performance isolation using currently available systems.

Cloud providers are aware of the problems of unpredictable performance of their clients. However, they continue to multiplex tenant services on their infrastructure for cost reasons. Cloud vendors usually allocate resources to their clients by apportioning available resources among them, using virtualization technologies. For resources like memory, tenants are generally more concerned about capacity than throughput and latency. This makes it easy for cloud vendors to multiplex multiple workloads with memory requirements. For other resources like CPU and network, time-sharing is efficient and causes only minor performance interference when shared across multiple tenant services. Cloud providers can easily allocate such resources as needed by the clients and can manage potential performance problems by tuning the degree of over-subscription of the resources. However, storage devices behave in a different manner. Unlike other resources, storage resources are not apportioned among clients using virtualization technologies. Instead, cloud providers offer their tenants a shared storage system in their datacenter. Such shared storage services are based on storage devices like Hard Disk Drives (or HDDs) or Solid State Disks (or SSDs). For these devices, throughput and latency are of much greater importance than capacity. Dividing storage device's resources in terms of capacity does not guarantee proportionate performance for the clients sharing the device.

Dividing these storage devices among clients, like other computing resources, i.e., based

on their capacity, severely hampers the overall performance of the device. Using traditional resource-sharing policies like time-sharing can significantly degrade their ability to meet performance requirements, such as latency deadlines. The primary reason for such performance behaviour of traditional storage devices like HDDs is the involvement of mechanical components like disk-arms. However, even if these HDDs are replaced by faster, non-mechanical Solid State Disks (or SSDs), the problem of performance interference remains. The performance of SSDs is still not comparable to main memory, and the storage device would still be the bottleneck in the system. Specifically, depending upon the workload, garbage collection in SSDs can be triggered at any point of time during the course of its operation. Since these garbage collection procedures adversely affect the I/O performance of the device, they can make performance characteristics of SSDs unpredictable. These issues restrict the type of applications which can deployed onto the cloud. For instance, in our example of an online shopping website, unpredictable performance from their storage back-end can lead to fluctuations in end-user latency, resulting in loss of revenue [28].

1.2 Taking a Middleware Approach

One possible approach to address the problem of performance interference is to incorporate tenant and deadline-aware request scheduling into existing cloud storage systems. However, there is a wide variety of cloud storage systems available today [43, 9, 5, 25, 8] each offering a different set of functionalities, optimizations and trade-offs. Each one of these systems focuses on a particular application-type and usage-scenario, and are designed to meet the very specific expectations of those applications from the data-storage system. For example, Cassandra [43] and Dynamo [25] are known to handle network partitions well. FoundationDB [4] is good for applications that require ACID transactions across the databases. Therefore, there exists a wide variety of cloud storage systems, each targeting a particular aspect of system performance.

Since the use-cases for these systems are different from each other, there is no one market leader in this space. Programmers need to consider various characteristics of applications and deployment-scenarios, like reliability, scalability, consistency requirements, type of data, access-patterns, etc to make the right choice of cloud data-stores. Also, given that each cloud storage system offers a different feature set and interface, they vary in their storage format, data distribution and query language, among other architectural differences. This makes the process of migrating data from one cloud provider to another non-trivial. This is referred to as 'cloud vendor lock-in problem' [47]. This effectively adds another layer of inflexibility in the process of choosing the right cloud storage system for an

organization. A scheduling mechanism must also take many other factors tied to individual cloud storage systems into account when making a scheduling decision. This reduces the feasibility of a general solution, as any one scheduler will not have good performance for all cloud storage systems and implementing it for all cloud storage systems requires exceeding amount of effort.

1.3 Current Approaches

Most of the current approaches for achieving application-level performance isolation have static assignment of resources, e.g. virtualization technologies [16, 41, 38]. Each application in a shared resource environment can specify their priorities on a particular set of resources. In previous approaches a particular application will always be preferred over another application for the allocation of resources. This approach is effective for providing resources to high priority applications but will leave the low priority ones to resource starvation, which may cause them to miss their stipulated SLOs . Such a rigid nature of static resource allocation inhibits the allocation scheme from efficiently accommodating any workload variation, leaving no choice to the cloud service provider but to over-provision resources to provide the promised peak-load capacity during periods of high-activity for all services simultaneously.

There is a significant amount of past work on providing performance isolation in shared storage systems but only a few of these systems are dynamic in the way they allocate resources to the applications [44]. Apart from these systems other approaches need either the workload specific tuning of the storage service [19], an accurate model of storage service's performance characteristics [56] or modifications to the underlying storage system [62, 19]. However, none of these is very desirable considering the variety of applications and storage systems in different cloud deployments.

1.4 Contributions

In this thesis, we introduce a middleware solution for the problem of performance interference in cloud storage systems, called MicroFuge. As a part of MicroFuge, we present a deadline-conscious scheduling and a caching layer for cloud storage systems, similar to the external distributed caching layers that are commonly used in most web service deployments [7, 55]. Using this additional middleware abstraction, we demonstrate that cloud storage systems are able to effectively provide much stronger performance isolation for

multi-tenant environments. As part of this thesis, we explore the effectiveness of making the distributed caching and scheduling system on top of cloud storage systems aware of application level SLOs.

MicroFuge provides performance isolation in part through a new cache eviction policy that is both tenant and deadline-aware. Unlike Memcached, where there is a single LRU queue for managing cache eviction, MicroFuge introduces multiple deadline-specific LRU queues for different ranges of deadlines, where items with shorter deadlines (or items in the earlier-deadline queue) are less likely to be selected for eviction than those with longer deadline (or items in the later-deadline queues). By increasing the cache hit-rate of shorter deadline items, MicroFuge increases the total percentage of deadlines that it can meet. This is because requests with shorter are more likely to exceed their SLOs, if they are not served from the cache. MicroFuge can also preferentially evict a specific tenant's data items if it detects a significant disparity between the memory capacity utilization of different tenants as a means to improve fairness. We focus on meeting deadlines rather than providing fairness in this thesis.

Although a caching policy that has knowledge of the workload can help to meet more performance requirements, it is nevertheless impossible to meet aggressive latency deadlines given an arbitrary request load. We address this problem in MicroFuge with the addition of an extra layer, the distributed scheduling layer, called Deadline Scheduler (or DLS). DLS controls access to the cloud storage system by scheduling requests to it in a deadline-aware manner. DLS's scheduling policies take scheduling decisions based upon the captured performance characteristics of storage accesses and helps requests to meet their response-time deadlines. In addition to this, DLS performs admission control to reject requests that are unlikely to meet their latency deadlines. The scheduler, with its underlying admission control mechanism, ensures that a certain number of requests (a tunable parameter) can still meet their performance requirements regardless of workload characteristics. Also, the applications making these requests can make informed decisions on their next course of action (like reducing workload or using some static content) if notified with early request rejections rather than knowing later that their request was not serviced by the storage system within the stipulated time-frame.

MicroFuge enables developers to specify shorter response time deadlines for 'must have' items, while longer or best-effort response time deadlines for 'nice-to-have' items. For instance, an online shopping website, if the storage system is lightly loaded, and all the items (both primary and auxiliary) can be fetched from the storage system within the response time threshold of the whole page to the end-customer, all the items are laid out on the page. On the other hand, if auxiliary items are likely to miss their deadlines from storage system, the front-end application would be notified (with reject-message) by

Micorfuge's API, which in turn can modify its page layout to show some static content (like non-relevant ads), avoiding the page from missing few sections. Therefore, the ability to specify the latency deadlines in MicroFuge API and to bound the deadline miss rate would help the website in keeping its response time (or page-load time) below the ever-decreasing [64] threshold of acceptable web-page response times for end-users.

The main contributions of this thesis are as follows:

- The design and implementation of MicroFuge, including a tenant and deadline-aware distributed caching system.
- A distributed scheduling system, also part of MicroFuge, that performs request admission control and ensures that requests can meet their performance requirements even when the system is heavily loaded.
- An evaluation that demonstrates the effectiveness of MicroFuge in a real deployment using the `YCSB` [23] workload.

1.5 Organization

The remainder of this thesis is structured as follows: Chapter 2 provides a background in to performance interference in shared storage systems, and discusses related work to the systems providing performance isolation in shared storage systems. Chapter 3 describes the design and implementation of the distributed caching layer of MicroFuge. Chapter 4 describes various design decisions and policies of the distributed scheduling layer of MicroFuge. Chapter 6 details our experimentation environment and workloads used, before presenting performance evaluation of every component of MicroFuge. Finally, Chapter 7 concludes the thesis.

Chapter 2

Background and Related Work

In this chapter, we provide the necessary background and examine current state of affairs in the field of performance isolation in cloud storage systems. We first describe the problem of performance isolation in shared storage systems. We then illustrate why taking a middleware approach to solve this problem works better and improves upon adoptability of such systems. Furthermore, we summarize different approaches for achieving performance isolation in traditional storage systems and discuss how they can be used in currently popular cloud storage systems. In the end, we explore current systems targeted to achieve performance isolation in cloud storage systems specifically.

In the context of this thesis, the target metric of performance is latency deadline. Since, response time SLOs and latency deadlines represent the same quantity here, we would be using these two terms interchangeably.

2.1 Performance Isolation in Shared Storage Systems

The cloud storage systems, both private or public (as in case of cloud datacenters), typically multiplex variety of tenants for cost effectiveness. Unlike dedicated shared storage systems, resources in a cloud storage system need not to be provisioned to accommodate peak-load requirements of each tenant. Cloud vendors take advantage of statistical multiplexing [20] to ensure proper utilization of the resources and cost-effectiveness for the customers. Cloud-based storage services often form the backbone of multi-tenant application environments in datacenters. Figure 2.1 illustrates how cloud resources can be shared among multiple tenants at different levels of abstraction.

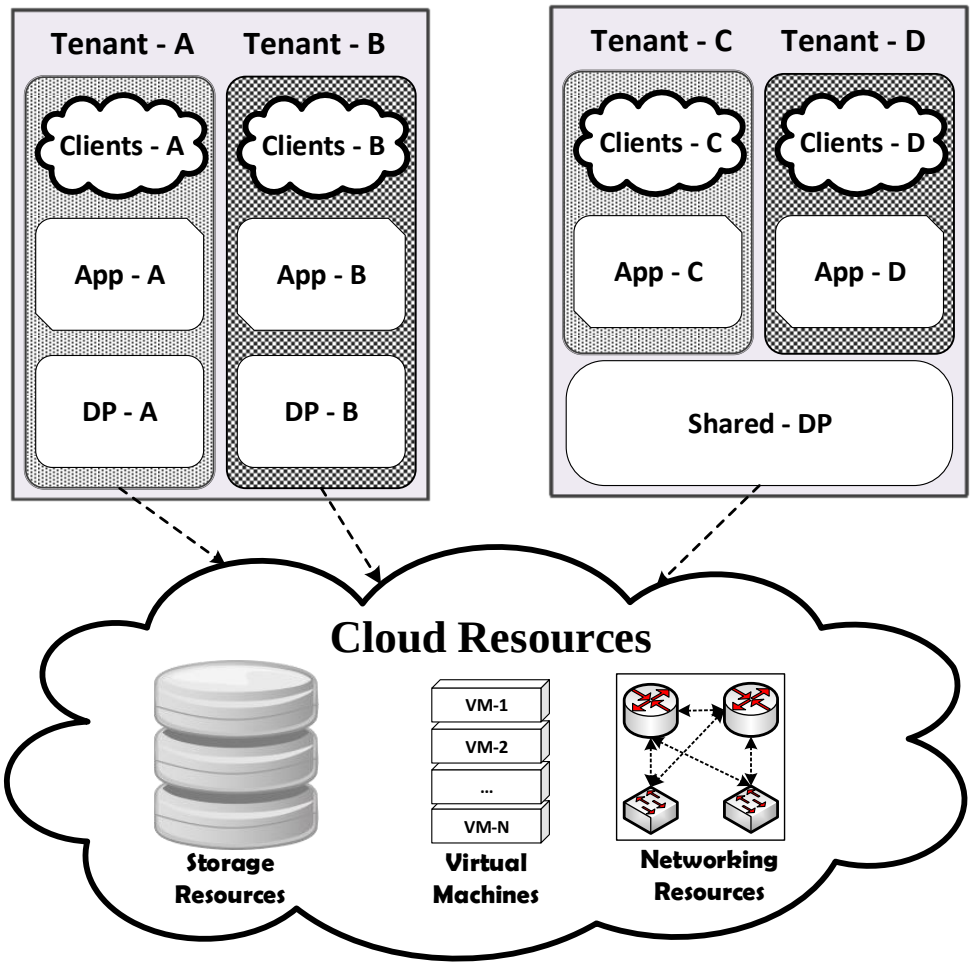


Figure 2.1: Mutli-tenant application environment. Tenant-A and Tenant-B shares the cloud resources at infrastructure level but have their own data-platform (represented as DP) and software application (represented as App). Tenant-C and Tenant-D have their own software applications but share the underlying data-platform service.

Depending upon the type of applications hosted in these multi-tenant environments, their associated performance requirements could be conflicting in nature. For instance, an analytical map-reduce application (e.g, click-stream analysis) would demand high overall throughput from the storage service, while an user-facing transactional application (e.g.

online shopping website), on the other hand, would require low response time from the service. These performance requirements are generally referred to as Service-Level Objectives (or SLOs). When applications with conflicting SLOs are hosted together on the same underlying infrastructure, these storage services often introduce performance interference.

Since the resources in cloud datacenters are usually over-subscribed, the performance of hosted applications is often affected by each other. As mentioned briefly in Chapter 1, providing performance isolation in multi-tenant environments is fairly simple for computing resources like CPU, memory and networking. But this is not the case for storage services, mainly because of the mechanical nature of the underlying hardware (explained in details in next section). This is especially true if the applications competing for resources have different characteristics and have contrary performance metrics.

The performance, as observed by tenants of such shared storage systems, fluctuates quite significantly and is unpredictable. This not only makes potential cloud customers wary about moving their services to cloud, but also makes it difficult for the cloud-providers to guaranty differential service- levels and charge accordingly. It is for this reason, that most of the large-scale organizations host critical or popular applications in dedicated in-house storage clusters.

Distributed cloud storage systems typically consists of various components like persistent storage, external caching layer, controllers and filesystem cache. To provide any performance guarantees for concurrent workloads in the shared storage system, all these components must be designed in a way to support performance isolation among the clients. Traditionally, multiplexing of workloads on shared storage resources is engineered by partitioning the resources statically. There has been some research [35, 33, 63, 59] on sharing the resources using statistical multiplexing. In such systems external schedulers are used to perform admission control and request scheduling as a way to provide performance isolation. Other shared storage QoS schemes have been proposed using time-slicing of resources. Matthew Wachs et al. [60] used synchronized global time-slice schedules to coordinate independent storage-servers' activities as a way to achieve performance isolation for dealing with stripped data.

2.1.1 Current Approaches for Performance Isolation

Most of the current approaches either modify one or more components of storage systems or add another layer of abstractions on top of these components. In this thesis, we solve this problem of achieving performance isolation in part by introducing an external caching layer and a distributed scheduling layer. We call our distributed memory based caching

system DeadLineCache (or DLC) and our deadline-conscious distributed scheduling layer Deadline Scheduler (or DLS). Both the components are deadline-aware in the way they function and helps in providing performance isolation in back-end cloud storage systems. We defer the discussion about other approaches for providing performance isolation using caching layer to the next section.

Previous works have examined cache sharing and scheduling for multi-tenant systems, and the Argon storage system [61] shares some similarities with Microfuge. Argon introduces a storage server that provides intelligent cache sharing between multi-tenant workloads, in addition to explicit workload isolation guaranteed by disk-head time slicing. The focus of Argon was not on meeting deadlines in the system, rather on how to use isolation to provide improved throughput and accurate exertion-based billing, where every user of the system is charged based on their usage patterns (ensuring that users with poor access patterns pay more than users who are efficiently using the disk). Argon focuses on disk-head time slicing and simple cache sharing, whereas Microfuge makes explicit considerations for deadline-based requests when scheduling through the use of collected latency metadata. Microfuge furthermore directly addresses performance isolation through its multiple deadline-oriented LRU queues, and its eviction policy that favors data with tighter deadlines.

Another similar work to our own is the Frosting system [62]. Frosting proposes a request scheduling layer on top of a distributed storage system. Like Microfuge, Frosting allows applications to specify high-level Service Level Objectives (SLOs), which are in turn automatically mapped into scheduler decisions. A feedback controller is employed to make scheduling decisions more predictable, and Frosting furthermore attempts to bound outstanding requests while minimizing queuing at the data store layer (in an effort to increase response times). Frosting, like Argon, focuses mainly on system throughput and fairness, not on strict performance isolation.

Pisces [57] is another system related to our own. Its creators propose a group of mechanics for partitioning resources between users. The authors suggest that by considering partition placement, weight allocation, replica selection and fair queueing for resources, the system can split aggregate throughput in the system between clients. Although Pisces does provide throughput isolation for performance, it does not provide beyond this: The system has no concept of deadlines and latency-bound tasks and does not attempt to perform isolation in this respect.

The Fast system [45] took a different approach to performance isolation for multiple cloud tenants, introducing a block-level replicated storage service that helps provide performance predictability by overlapping similar type of operations (sequential reads or random

writes, for example) on the same machines, minizming interference. Unlike Microfuge, Fast's primary focus is on system fairness, and it additionally does not consider request deadlines.

In the Basil system [34], a scheduler automatically manages virtual disk placement and performs load balancing across physical devices, without assuming any underlying storage array support. Load-balancing in the system is based around I/O load, and not simply data volume. While this emphasizes the need to control access to data stores, it does not actually focus on cloud data stores, and does not provide the type of client-specified, deadline-oriented service that Microfuge does.

Similar to the ticket-based reservation system used by Microfuge's scheduling layer, SQLVM [51] proposed using both resource reservations and metering techniques to help share resources between clients in a multi-tenant database environment. SQLVM is not implemented as middleware for distributed storage systems, but rather as an environment for multiple DBMS systems running on the same physical machine. It furthermore does not allow client-specified deadline requirements for requests.

As the popularity of cloud-based applications has increased, several key-value stores have been proposed to provide enhanced performance over relational database systems by relaxing ACID properties [43, 9, 5]. These key-value stores, designed with the cloud in mind, can often suffer from poor performance isolation. This can lead to contention for shared physical resources. In particular, hard drives can become very heavily loaded in cloud storage systems.

There are many approaches to performing both external and internal scheduling for admission control, and we focus on several which can be seen as inspirational for our thesis. Schroeder et al. [48] considers optimizing concurrency levels in database systems through admission control. Abbott and Garcia-Molina [12] propose models for performing admission control aimed at real-time database systems using deadlines. They use simulations to understand the performance trade-offs of utilizing transactional commit behavior for admission control. Microfuge makes use of admission control and scheduling in order to provide multiple tenants using the same storage system with performance isolation. Our system ensures that a certain subset of all requests with client-provided access deadlines can still be completed regardless of system load.

2.2 Distributed Caching Systems

Distributed caching systems are memory based caching systems which are typically used as temporary and fast storage for frequently or recently accessed data. These caching systems are generally much more efficient than the origin service or system, and therefore, are very effective in improving the performance of the overall system. Also, since the caching layer off-loads the origin service, it is very helpful in making the overall system more scalable, flexible and fault-tolerant.

2.2.1 Types of Distributed Cache Systems

Caching systems can be divided in to 2 categories [58] :

- **Forward-proxy Cache Systems**
  Forward-proxy cache systems can be distributed in various parts of the infrastructure between the client and the server, like in dedicated traffic servers [2] or in browser cache. Common examples of forward-proxy caches include Content Delivery Networks (CDNs) [29] for storing static content and web-cache for caching dynamic web-pages of a compute-intensive web-service, like web-search engines (e.g., Google and Bing).

- **Reverse-proxy Cache Systems**
  Reverse-proxy cache systems, on the other hand, are placed close to the back-end service infrastructure and more often than not are installed in the same server farm. Web accelerator caching systems like Squid [10] and Varnish [11] are one of the most commonly used reverse-proxy caching systems.

However, the purpose of these two types of cache systems is quite dissimilar : Forward-proxy systems are mainly used to keep the data close to the end-users as it reduces their effective latency. While reverse-proxy systems, on the other hand, are about offloading the origin servers and increasing the efficiency of the back-end service or system, like a Database system or DNS.

In this thesis, we focus on caching systems which are used to cache data from the back-end database systems. These caching systems are often placed like reverse-proxy caches in application architecture and are generally referred to as Database Caching systems. However, depending upon the implementation they may or may not act as proxy for the

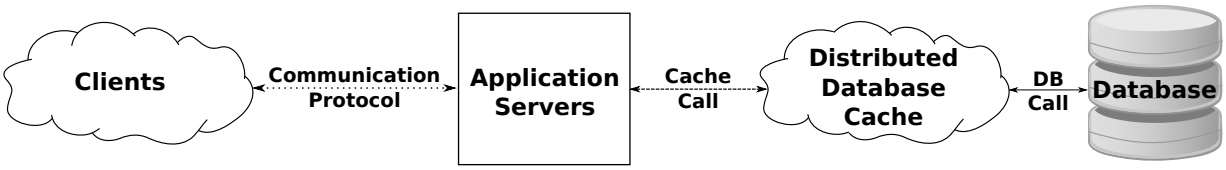


Figure 2.2: Middle-tier Distributed Database Cache

origin database servers. Examples of such systems include Memcached [7] and Windows Azure Caching [52].

Memcached [7] is the most popular open-source implementation of distributed memory object caching system. The most common use-case of memcached is as a database caching layer. All the large scale websites like Facebook [54], Youtube [24], etc use memcached or other similar caching systems for performance and scalability reasons. For example, Twitter uses their own version of heavily modified memcached, called Twemcache [14], for caching of Tweets. While 'tweets' are persistently stored in the disks, Twemcache caches most recently and frequently accessed tweets, alleviating the load on the disk based database system. Also, Twitter uses Twemcache to cache tweets formatted in a particular format (say, tailored for a cellphone application), to save repeated data-rendering overhead for application server's CPU. Using such caching system not only helps Twitter to make their product more scalable but it also helps them in meeting their response time SLOs.

2.2.2 Advantages of Middle-tier Caching

Database caching system are usually deployed in a middle-tier of a multi-tier application and are typically shared among multiple storage servers. These shared caching systems may not outperform the dedicated local application-specific caches, as these generic caching systems are not aware of specific caching requirements of the application. However, like other distributed caching systems, they have the advantage of being collaborative, which means that each application hosted on the cache has access to more cache capacity than if the memory was used for a local application-specific cache.

Adding this additional layer of caching in any database application, improves following aspects of the overall system [46] :

- **Scalability**
  These caching systems reduce the back-end database servers' load by distributing

the query workload to multiple front-end caching servers. Along with this, since these cache systems simulate a big hash-table, with almost no coordination among its nodes, they are horizontally scalable. Therefore, adding such a caching layer on top of the database makes the whole system more scalable.

- **Flexibility**
  As mentioned before, these middle-tier caching systems gives more flexibility to administrators of the shared services, like database system, to provide differentiated services to the clients. For example, in the event of shared cache reaching its capacity, data of low priority customers will be evicted before the data of high priority customer.

- **Performance**
  Since the cache is multiple orders of magnitude faster than back-end database systems [31], the average response time of the overall system improves drastically, even with modest cache hit-rate. Moreover, these systems take advantage of locality-patterns in the workloads and make the load to the back-end system more uniform, helping them to meet client specified response time SLOs [27].

- **Availability**
  These caching systems can also sometimes be useful in marginally improving the availability of the back-end systems. For example, if the back-end service crashes, cache can still serve the clients' requests, provided that clients ask for only those data items that are already present in the cache system.

In this work, we specifically focus on how database caching systems can be utilized to achieve performance isolation in multi-tenant cloud storage systems.

2.2.3 Caching Policies

Laszlo Belady [17] in 1966, proved that the most efficient caching algorithm would be the one which evicts the data items that are least likely to be needed for the longest time in the future. However, such a policy would be unworkable without application-specific knowledge or the knowledge of workload characteristics beforehand. Least Recently Used (or LRU) based cache eviction policy, where items are ordered and evicted by their last access time-stamp in a queue, approximates the above mentioned perfect caching algorithm for slowly varying probabilities, without requiring to estimate those probabilities [13]. Chrobak et al.[21] proved that the superiority of LRU over other generic cache eviction policy like FIFO

or LFU can be attributed to the locality of reference phenomenon exhibited by request sequences in common cache workloads. Later, O'Neil et al.[53] showed that LRU-based cache eviction offers near-optimal caching performance. Therefore, for generic caching systems, where cache does not have any domain knowledge, LRU [7] and even probabilistic LRU [3] based eviction policy works quite well.

2.2.4 Example Deployment Scenario

In this section, we expand on the example of an online shopping website selling large variety of commodities. We will illustrate how database caching can help the administrators of the website to better meet the application level response time SLOs. Figure 2.3 represents a typical multi-tier architecture for a large-scale website.

Each commodity sold by the business has its own web-page, which provides the details of the product. For modern dynamic websites, web-servers commonly have to dynamically pull the contents of a web-page from a number of internal sub-systems or services. These services can potentially be shared with other applications of the enterprise, as shown in Figure 2.3. Examples of such services can be a service providing news feed of the related items, or a named entity recognition engine and so on. For the core content of the page, data is first checked in Database Cache, if not found, it is fetched from the back-end datastore.

Queries to these back-end services are resource-intensive and are expensive in terms of latencies. Therefore, if we store these rendered web-pages in cache, the whole process of serving the formatted content is accelerated. This is shown as Caching Proxy System at layer-3 in Figure 2.3. To avoid staleness of the content, application should keep the cache updated, each time any attribute of the web-page gets changed. All the Internet giants like Amazon, Facebook and Google use such caching solutions for reasons mentioned in previous sub-section (see Section 2.2.2).

2.2.5 Current Research in Caching Systems

External caching layer plays an important role in improving the performance of shared storage systems. As a side-effect of improved performance, they are helpful in achieving performance isolation in these cloud storage systems. A workload and deadline-aware database caching system, can assist performance isolation between concurrent clients, leading to invariable and predictable performance. However, there is very limited past work on using the caching layer to provide performance isolation in back-end cloud storage systems.

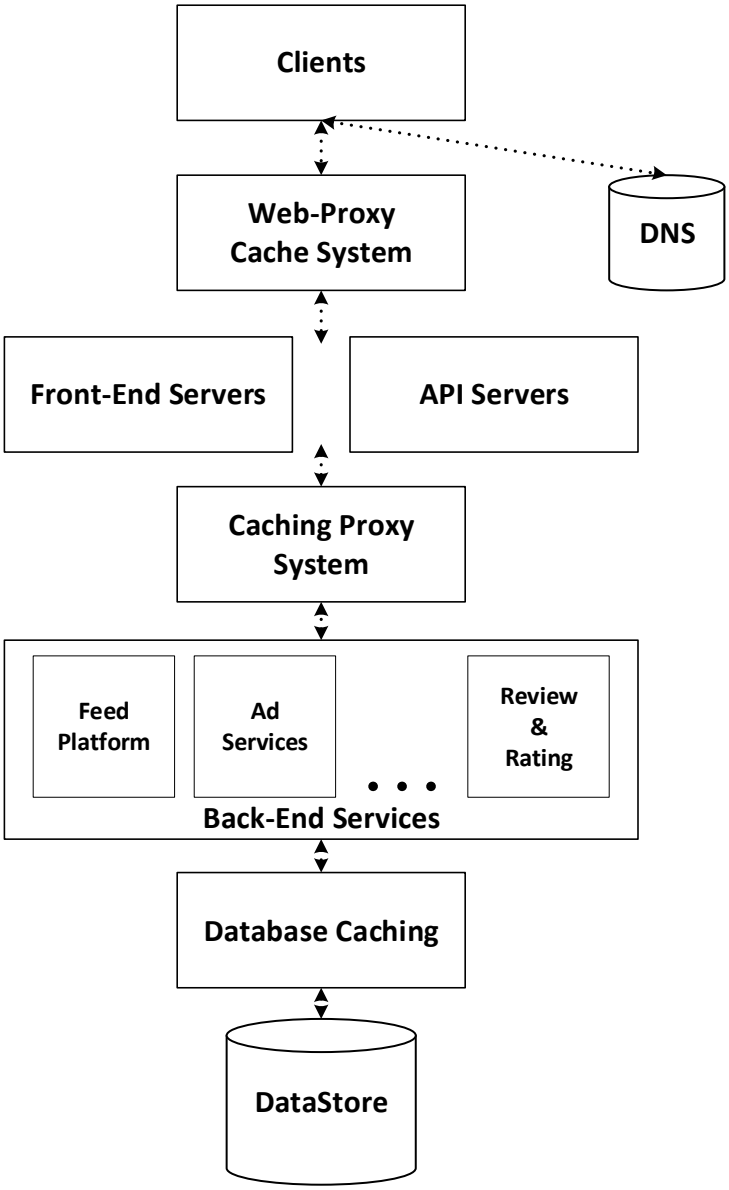


Figure 2.3: Simplified architecture of a large-scale website

Memcached [7] is a general-purpose distributed memory caching system, which is mainly used to provide a scalable memory-based caching layer for data stores, thereby improving access latencies. As a simple external cache application, however, Memcached does not provide performance isolation. Additionally, unlike Microfuge, Memcached does not perform deadline-aware caching and scheduling. As a result it also does not have any provision for admission control.

MemC3 [26] proposes to use optimistic hashing with CLOCK-based cache management in Memcached to improve on the access latencies in Memcached. Despite further improvements to access times for items resulting from the more advanced cache eviction algorithm, the system still does not provide the performance isolation that Microfuge offers through the use of deadline-based caching and scheduling.

Nahanni similary modified Memcached to provide a inter-VM shared memory [32], with a good degree of success and the potential for complementary usage of its presented techniques with other caching systems, but was limited to VMs running on the same physical host and provided almost no performance isolation guarantees.

Chapter 3

Deadline Cache

Distributed caching plays a vital role in improving the performance of cloud storage systems. Most current caching schemes are designed to improve performance and reduce the load of back-end storage servers. As a side-effect, by improving overall performance, these caching systems also reduce the average access latencies of cloud storage systems and enable them to meet more client specified SLOs. Current caching systems are oblivious of these application-level requirements. In this chapter, we introduce a new distributed caching layer called Deadline Cache (or DLC), which is deadline-aware and improves the performance isolation in cloud storage systems.

DLC aims to improve the performance isolation in cloud storage systems by selectively caching the requested data. Specifically, data items that have shorter response time requirements are less likely to be evicted from DLC compared to the items having longer response time requirements. This is based on the assumption that requests with longer deadlines will meet their deadlines even if they are served from the cloud storage system. Additionally, Deadline Cache can also distinguish between different tenants to make sure that there is fairness among them.

3.1 Deadline Cache Architecture

In this section, we introduce a new deadline-aware distributed caching system called Deadline Cache (or DLC), with the aim to improve performance isolation in cloud storage systems. DLC preferentially keeps the data items with shorter response time deadlines using its deadline-aware cache space architecture and eviction policy. DLC provides the

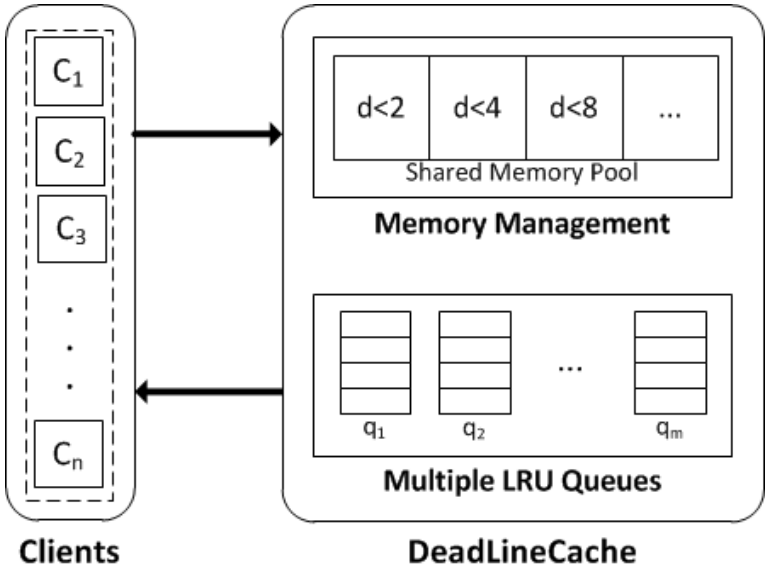


Figure 3.1: DLC architecture. Multiple Clients contact the distributed caching layer for items with different response time requirements.

same API as Memcached, and is easy to layer over top of almost any cloud storage system. A visual representation of the DLC architecture is presented in Figure 3.1.

The three primary elements of Deadline Cache that allow it to perform deadline-aware caching are:

- Multiple Least-Recently Used (LRU) queues, where each queue is responsible for storing data having deadlines in a specific range, enabling a deadline-sensitive cache eviction policy.
- A cache management policy that takes deadlines and deadline-miss costs into account.
- A statistics collection system that enforces fairness and prevents applications from hoarding cache usage.

All these modules work in synchrony to provide performance isolation in the cloud storage stack, which in the context of this work is defined as the clients' ability to meet the application-specified response time deadlines, irrespective of other concurrent workloads on the shared storage system. The rest of this section describes how each of these module

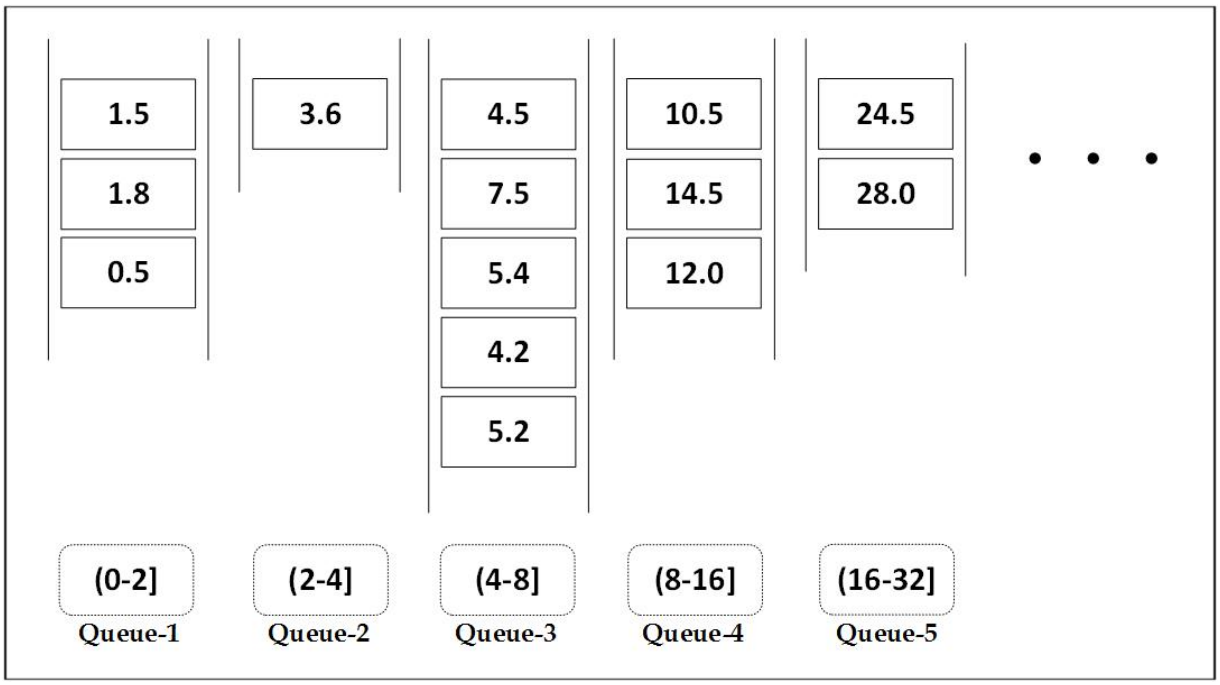


Figure 3.2: Multiple LRU queues in DLC to enable deadline-aware cache eviction policy. Each queue is responsible for items in a specific range of deadlines, and number in each data item represents its deadline. The range of each queue is shown in the box below it.

function and contribute in providing performance isolation in cloud storage systems as a whole.

3.1.1 Multiple LRU Queues

Although LRU achieves high cache hit-rate and good performance (see Section 2.2), it does not take request's response time deadlines into account when scheduling requests. The distributed caching layer is unaware of applications' high level SLOs [7] and most of the current solutions require tuning the parameters of the system until the desired performance is achieved [18]. Since performance of a tenant on a shared storage infrastructure depends upon multiple factors like workload characteristics and other concurrent clients, the process of tuning is non-trivial. In this section, we will analyze how making the caching layer deadlines-aware can help us in increasing the predictability of the performance of concurrent workloads.

In a system with strict performance requirements, the cost of a cache miss is substantially higher if it leads to a deadline miss, and the likelihood of a deadline miss is much higher for requests with short deadlines than those with long deadlines. To minimize the cost of a cache miss, DLC uses multiple LRU-ordered queues to determine cache item eviction ordering (as shown in Figure 3.2). Each queue is responsible for maintaining the LRU-order of items within a particular deadline range. In our current design, the size of each deadline range grows exponentially, with the exponential factor of 2. For example, queue 0 is responsible for items with deadlines less than 2 ms, queue 1 is for deadlines between 2 to 4 ms, and queue 2 is for deadlines between 4 to 8 ms, and so on.

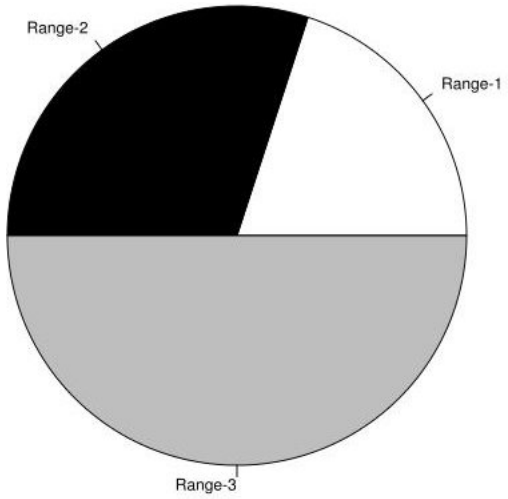


Figure 3.3: Deadline distribution in a workload. The sectors in this chart represents different type of requests having different range of response time requirements from storage system. With in these sectors, distribution is considered to be uniform for the purpose of our experiments.

The exponential distribution of deadline-ranges works well for majority of workloads having wide variety of deadlines. Such a deadline-distribution in a workload is representative of real-world storage workload characteristics, where a set of applications have limited variety of requests going to the shared storage system, and with in this spectrum of requests, deadlines are uniformly distributed. For instance, in our running example of an online shopping website, only 3 types of request goes to storage system : product (or feature) search request, request for product-details and related-item search. Among these, product search type of request would have the lowest deadline (or highest priority) requirements, product details request would have intermediate deadline requirements, while

related-items search would be fine with high deadline requirements. Depending upon the desired granularity in expected response time, with in these broad ranges of deadlines, there may be some request which are more urgent than others and the other way round. For example, a shopping site may want the data pertaining to a product with high margin of profit to be serviced with higher priority as compared to data related to low profit margin product.

DLC does not partition the cache space statically. Depending upon the distribution of deadlines of requests in the workload, the queues in DLC grow or shrink. The size of the queue is representative of proportion of cache-space taken by data items of requests belonging to a specific range of deadlines. The overall objective of DLC is to give priority to data items of requests having urgent or smaller deadlines as compared to data items of requests with uncritical or larger deadlines. This is achieved by deadline-aware cache eviction policy of DLC (see Section 3.1.2), and such a multi-queue architecture is necessary for an efficient implementation of our cache eviction policy.

Figure 3.2 represents a snapshot of LRU-queues in one particular state of the cache system during the course of its operation. As can be seen, there is no static partitioning of the cache space. The fraction of shared memory-pool apportioned to a particular queue (deadline-range) depends on, the deadline-distribution of the workload and the cache eviction policy. For example, in Figure 3.2 Queue-2 has only one data item, while on the other hand, Queue-3 has 5 items in the queue and can grow even more if deadline-distribution of the workload happens to have more deadlines in the range of Queue-3, which is 4-8 ms in this case.

3.1.2 Deadline-Aware Cache Eviction Policy

In the standard LRU-based cache eviction, items are evicted from the head of the LRU-ordered queue. However, in DLC, to incorporate the deadline-miss cost into our eviction policy, we apply a deadline-based multiplier to the time difference between the current time and the LRU timestamp for each item. This determines the modified recency or effective age of an item. To limit overhead in computing this value, we select a fairly coarse-grained multiplier function that applies a linear multiplier to an exponentially growing range of deadlines. This linear multiplication factor is inversely proportional to the queue number of the corresponding queue (or bucket) to which a particular cache-item belongs, where queue number increases with increasing range of deadlines. For example, in Figure 3.2, for deadlines less than 2 ms, the multiplier is 1; for deadlines between 2 to 4 ms, the multiplier is 0.5; and for deadlines between 4 to 8ms, the multiplier is 0.25.

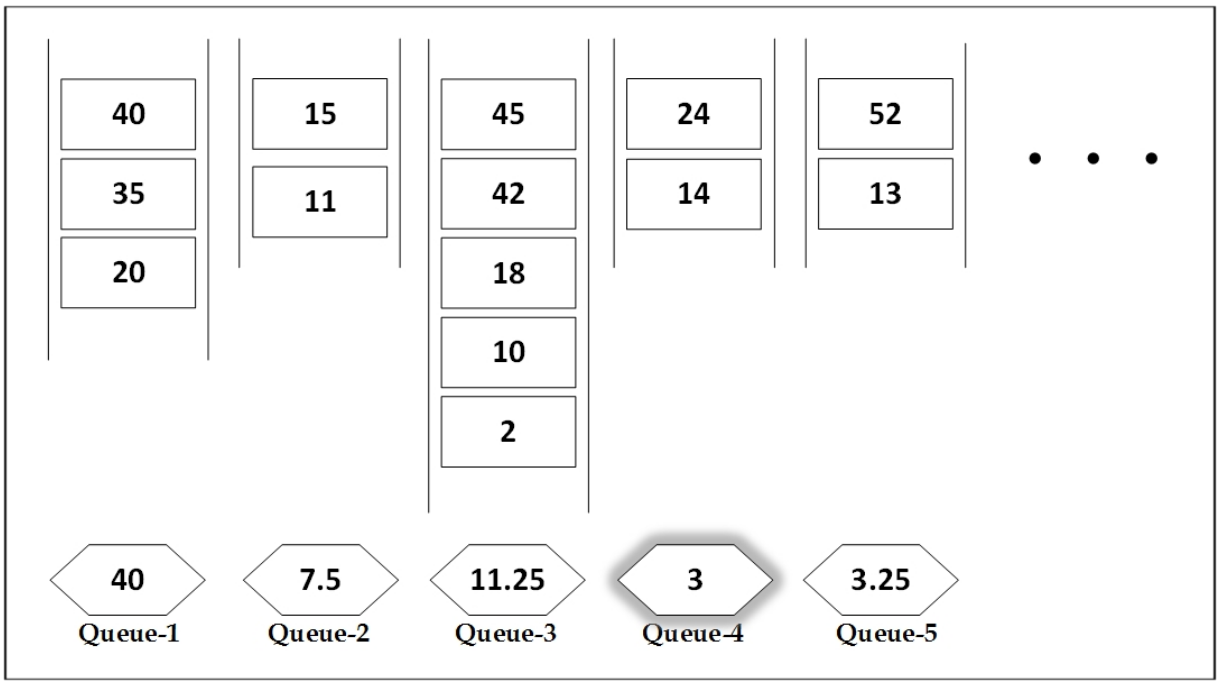


Figure 3.4: Deadline-aware eviction policy in DLC.

The relationship between the recency and the modified recency of an item is as follows:

$$R_w(item) = R(item)/c^q$$

where, R_w is the modified recency of the *item*, c is the constant factor in the exponential division of deadlines in to ranges, q is the queue of the *item* (as per its deadline range) and R represents the actual recency (or age) of the item in its own bucket.

Our deadline-aware eviction policy requires applying the deadline-based multiplier for items with different deadlines, and determines the oldest items using the modified recency metric. To perform these operations efficiently, we use the multi-queue data structure described in Section 3.1.1. Specifically, $modified - recency$ is calculated for head item of each of the queues (using the above formula), and among these head items, the one with least $modified - recency$ is chosen for eviction. This way we evict the item with maximum $effective - age$ from the cache, as these queues are in LRU-order and their heads represent the least recently used item in them. Given the exponential growth of the deadline ranges,

each eviction operation therefore requires reading from only a small number of queues, restricting the complexity of the eviction operation.

Deadline Cache's eviction policy is depicted in Figure 3.4. As mentioned before in subsection 3.1.1, each queue is responsible for data items in a specific range of deadlines, and number in each data item represents recency value of the item. The hexagonal box under each queue represents the $modified-recency$ (R_w) of their head element. After computing R_w of each queue's head, the head-item having minimum R_w (highlighted hexagonal box in figure) is evicted from the cache.

3.1.3 Fairness in the Caching Layer

To prevent a client from taking unfair advantage of the cache's capacity, DLC keeps track of the proportion of cache capacity used by each client and adjusts its eviction policy accordingly. At the time of eviction from the cache, in addition to the bucket (or queue) number of the data item (refer to Section 3.1.2) being used to determine the modified recency-value (R_w) of the item, the client's cache capacity usage is also considered to determine which cache item to evict. Effectively, closer the client is to its cache-capacity quota in terms of cache-space usage, higher the chances are of its data to get evicted from the cache. The exact formula used to determine the eviction item is:

$$R_w(item) = c^b * R(item) * \mu_{C_i}$$

where, μ_C is client C_i's cache capacity usage factor. The value of μ_{C_i} varies from 0 to 1. A high value for a client's capacity usage factor denotes that the client is occupying a large proportion of the caching system's memory and, in the interest of fairness, its item should be subject to eviction with a higher priority during periods of cache contention. It is to be noted that, cache-space is not necessarily partitioned equally among the clients and their portion may depend upon their service-level from the provider. This is a common way to provide Quality-of-service (QoS) in terms of capacity in shared infrastructure.

This feature has not been implemented, but would be necessary to provide fairness among clients, which is commonly associated with performance isolation in multi-tenant environment.

3.1.4 DLC API

DeadlineCache has a simple CRUD API [37] like other key-value pair systems, .e.g. Memcached [7]. In addition to regular arguments in primitive operations, Deadline Cache also

provides an additional argument to specify the response time SLO metadata for the data item. For example, in *put* operation a client can provide a hint of deadline of the data item for future fetches. Similarly, in *get* operation a client can specify the response time SLO for a particular key.

Specifically, the Deadline Cache API looks like :

- `get(key, deadline);`
- `put(key, value, overwrite, deadline);`
- `erase(key);`

This response time metadata is used by Microfuge, both Deadline Cache and Deadline Scheduler (see Section 4.2), to decide the priority of the data item while evicting items from the full cache and in the process of admission control, respectively. As mentioned before, a data item with shorter deadline is likely to stay longer in the cache, so that it can meet the response time SLO, which it may miss if served from the back-end system.

Chapter 4

Deadline Scheduler

The second major component of MicroFuge is its distributed scheduling layer, the Deadline Scheduler (DLS), which controls ordering and overall admission for requests to the cloud storage system in the back-end. Similar to our caching layer (Deadline Cache), the scheduling layer functions in a deadline-aware manner. As the main focus of this work is on providing performance-isolation in cloud storage systems, this layer aims to reduce the fraction of SLOs that are missed by the cloud storage system.

In a latency-critical shared system, it is necessary to have predictable performance. As discussed in Chapter 2, achieving predictable performance is especially difficult for shared storage systems. To make performance more predictable in these systems, we need to reduce the fraction of the requests that miss their deadlines. As discussed briefly in Chapter 1, we need admission-control to achieve this. Admission control in the context of this thesis would translate to rejecting the requests to the storage system as quickly as possible and notifying the client in case it is not possible for the storage system to meet the deadline for a request. By notifying the client that most probably the request is going to miss the response time SLO, Deadline Scheduler enables the clients to take some preventive measure rather than hoping for the response to come back in the stipulated time-frame. For instance, in our prior example of an online shopping website, the website company can serve some static content, like site-wide popular products, rather than related products on a product page if it gets a reject notification for related products query from the storage system. This can help the company in moderating the page-load time within the threshold of Tolerable Wait Time (TWT) of web-users [50] rather than letting the related-products module, which is beneficial but not critical to the business, extend the overall page-load time.

Deadline Scheduler has three main components in its architecture. These smaller components work in unison to provide the scheduler with the capability to provide performance isolation for the back-end storage system. These elements are:

- A ticket-based ordering system that controls client access to the storage layer.
- Performance statistics collection and admission control that rejects requests that will likely miss their deadlines.
- A system that prevents clients from monopolizing storage layer resources, further enforcing performance isolation.

This chapter outlines the functionality of the different elements of the scheduling layer that operate on the scheduler nodes. A depiction of clients interacting with the scheduling layer and the data store can be seen in Figure 4.1.

4.1 Scheduler Ticket System

Request ordering functionality of our Deadline Scheduler is build over a ticket based system. A scheduler server is responsible for tracking all the requests going to one or multiple data-servers (individual storage servers in a distributed storage system). This enables Deadline Scheduler to do centralized scheduling for requests going to each data-server. A client that wishes to have its requests serviced by the cloud storage system initiates request(s) to one or more DLS servers to schedule the requests on the data server(s) containing the data item. In parallel to sending ticket requests to DLS, the client also sends a query request to the Deadline Cache. It then waits for a response from Deadline Cache, determining whether or not the requested data can be served from the cache. In case of a cache hit, the client would cancel the scheduler-tickets. On the other hand, if it is a cache miss, based on the scheduling tickets received from the Deadline Scheduler, the client dispatches its request to a particular data-store server of the cloud storage system. We now discuss the exact order of messages between a client, Microfuge (including both Deadline Scheduler and Deadline Cache) and the underlying cloud storage system in different scenarios.

A typical storage request starts with determining the list (which depends upon the configured replication factor of the storage system) of the data-server nodes that contain the requested data. This can be found out using the client library of the back-end cloud storage system. Depending upon the architecture of the cloud storage system, the key

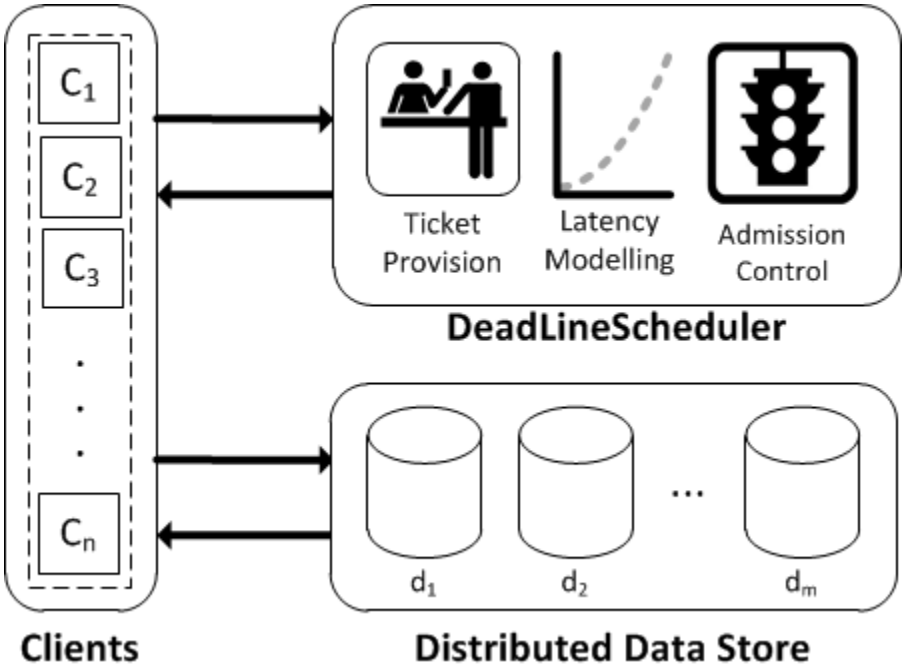


Figure 4.1: DLS architecture. Clients contact the distributed scheduling layer to acquire tickets. If their deadlines are not rejected they are provided a queuing ticket. After returning from the data store, clients pass latency information to the scheduler.

to data-server mapping can be either found out from a central directory server (as in the case of Google File System [30] or HBase [5] and PNUTS [22]) or from Hashing [40] (as in the case of Amazon's Dynamo [25] and Cassandra [43]). After determining the set of corresponding data-servers, a client randomly selects two of the nodes [49] and sends their corresponding schedulers a ticket request that includes the deadline requirements for the client. Each of the schedulers independently determines whether or not the incoming request would violate its performance requirements (discussed in depth in Section 4.2), and either informs the client that the request could not be performed within the required SLO or the other way round, in which case the client would successfully acquire a ticket. From clients' perspective, acquiring a scheduler-ticket for a storage node guarantees that the request will meet its deadline if served from this particular node in the cloud storage system.

If client receives a rejection message from both the scheduler servers, the request is effectively cancelled. If it receives only one successful ticket it blocks on it. If both the

tickets are accepted by the schedulers, client will choose the ticket (or data server) with shorter response time and cancel the other one. Chapter 5.1 talks about the procedure of selecting a ticket in more details.

Clients can also tag their requests as 'best-effort' meaning the application requires delivery or storage of data without any expectation of response time. In this case, the scheduler always accepts the scheduling request and returns the expected response time for the data item. Even though these requests are best-effort, once accepted by scheduler their priority is not changed even to accommodate requests which are not best-effort. This makes our API more reasonable for application developers. Since in the case of best-effort requests, both scheduling tickets will always be accepted, the ticket (or data-server) with shorter response time would be chosen for serving the request.

Once our client library decides which data-server is going to serve the request, it then waits on a response from the scheduler. This wait-request remains blocked, waiting on its ticket until all accepted requests with earlier or shorter absolute deadlines (according to the scheduler's determined ordering) are completed. Only after getting the response from the scheduler the client will proceed to contact the particular data store server.

The scheduler is implemented over a simple queue, and services requests according to a modified shortest deadline first policy. Typical shortest deadline first policies can lead to starvation for requests with long-lived deadlines (if they are continually pushed down in the queue by requests with short-lived deadlines). However, this type of starvation does not occur in DLS as its scheduling policy restrict the number of times the priority of an already queued request can be decreased. It also provides its own admission control mechanism and rejects incoming requests that would cause deadline violations for already-queued requests. Therefore, a request which has been already accepted by the scheduler has a very small chance of missing the stipulated deadline unless there is some serious hiccup in the system.

As mentioned above, requests that have successfully received a ticket are not removed by the scheduler until they are serviced (and thus cannot be rejected once issued a ticket). After accessing the data store, the client sends parallel requests to update the cache with the new data-item and releases its ticket from the scheduler while providing the scheduling server with the newly obtained request latency metadata.

In practice, this scheduling algorithm is highly efficient, and performs very well under a wide variety of conditions. Our implementation builds on the techniques determined by Mitzenmacher [49] in 2000, according to which the design of choosing the shortest of the two randomly chosen replicas' queue performs well under a large range of system parameters. This makes our scheduling policy significantly better than other similar scheduling methods as we avoid using global information and introduction of additional overhead

and complexity, which provides marginal performance gain. As mentioned by Amazon in its Dynamo paper [25], the cost of improving performance increases significantly when a storage system attempts to keep the percentage of SLOs missed in single digit. In such a situation, admission control is the only way to bound the SLO misses by the storage layer, without incurring the extensive cost of increasing the capacity of the storage system.

4.2 Request Admission Control

The MicroFuge system is designed to prevent storage requests from missing their response time deadlines. Deadline-awareness in the cache eviction policy helps reduce the likelihood of deadline misses by increasing the cost of evicting items with short deadlines. The ticket-based scheduling algorithm in our scheduling layer helps distribute load and, incidentally, increases throughput by preventing interleaved seeking on the storage system (as it only deals with requests from a single source at a time). Despite all this, as the load on the system increases, deadline misses are inevitable and previous work has shown that outstanding requests on a data store can significantly impact response times [34]. Therefore, to satisfy the performance requirements of at least a subset of the requests, new requests must be rejected by an admission control system.

As requests complete and request the scheduling layer to remove their tickets, they provide the scheduling node with latency information that details how long the request took to complete. The MicroFuge system keeps track of the latency information of how long each data-server is taking to service a request. It is to be noted that this latency information does not consider the time taken to acquire a ticket from a scheduler node or to remove the ticket from the scheduler node after the request is serviced. When notifying the scheduler that the request corresponding to a ticket has been serviced and the ticket should be removed form the queue, this latency data is included with the message. DLS in turn uses the past response time information to generate estimates of how long an incoming request should take to get the response from the storage system including the queuing delay on Deadline Scheduler.

When clients make requests with deadlines that cannot be served from cache and will likely not succeed in meeting its deadline as per the response time estimates from the storage layer, the scheduler notifies the client that its ticket-request has been rejected. Additionally, requests may also be rejected if adding them to the scheduler will likely prevent other already-accepted requests from being able to complete within their respective deadlines.

By allowing applications to define their own deadlines on storage requests at the granularity of each request, MicroFuge allows for the hosting of applications that cater to clients with a variety of latency requirements. In our design, performance of a request is governed by client-defined performance metrics, and not the generic (and possibly not so helpful) decisions of the scheduler incorporated inside the data store. It furthermore empowers clients to know almost immediately if a request will not be serviceable within desired time limits, allowing clients to react quickly and effectively when I/O deadlines cannot be met.

Such an admission control is also useful when we want to shrink the resources or infrastructure allocated to the application, during the period of low-activity. For example, the load on a website could depend on the time of the day and day of the week. During the period of low-activity like nights or weekends, these organizations tend to decrease the resources allocated to their web infrastructure. for cost reasons. However, there is no direct translation of decreased load (requests/second) to amount of resources required to achieve the target performance. In such a scenario, we can have another component which can monitor the rejection rate from our scheduler and adjust the amount of resources allocated to the application. This will enable the organization to save costs without losing application performance.

4.3 Fairness in the Scheduling Layer

The overall goal of Deadline Scheduler is to provide a system that can meet application-level response time requirements through application-specific scheduling strategies to maximize performance for clients based on their individual needs and scheduling goals. Unfortunately, a naive version of this system would be vulnerable to greedy or malicious clients that repeatedly issues many requests with low deadline requirements. The naive version also fails to allow resource provisioning based on quality of service guarantees.

The scheduling layer of MicroFuge deals with these issues by making scheduling QoS-aware in addition to being deadline-aware. The system supports the definition of mutliple different service levels, and can use these levels to provide relative fairness based on QoS guarantees. The system can, for example, bound the proportion of requests that are rejected for clients belonging to various service levels, ensuring that clients belonging to a lower service level (any client in general) cannot monopolize the resources of the system. In these cases, the scheduler will simply reject greedy ticket requests that violate QoS policies.

Allowing for explicit quality of service definitions is not enough to solve all fairness issues that arise in a scheduling system. In particular, the deadline-focused nature of DLC

makes it naturally biased against requests with shorter deadlines during periods of high contention. In these kinds of cases, without appropriate intervention, it would be possible for a heavily loaded system to always reject requests with lower, hard to meet deadlines. To rectify this, the MicroFuge client re-issues rejected requests at low rates. These requests will always be accepted, and when the system eventually caches them, they are inserted into DLC with their original deadline information.

Although fairness is important in practice in various other settings, it is not the focus of this work. Hence, this work does not present experimental evaluation of the system's fairness mechanisms but rather chooses to keep the focus on performance isolation.

Chapter 5

MicroFuge Protocol

MicroFuge uses a specific protocol to push requests through the system. This chapter describes the MicroFuge request protocol, and then outlines the beginning-to-end progress of a sample read request.

5.1 Protocol Details

Upon receiving a request, clients that wish to take advantage of the cache layer contact a cache server to determine whether or not the request can be serviced immediately from cache. When a request is serviced directly from the cache, the client is informed of the requested value, and the protocol terminates. Requests that are not serviceable from cache are allowed to continue through the scheduling layer.

Should the request not be serviceable from the cache, clients simultaneously begin communication with two randomly selected nodes (servers) from the scheduling layer, both of which are responsible for a data server node which holds a replicated copy of the data in the request. The client informs both nodes of the request, its deadline and whether the deadline is a hard deadline or whether the request is to be made in a best-effort manner. Currently, these scheduling nodes are determined through simple hashing, but could equally be determined by contacting a central directory server (like namenodes) or through other strategies.

A client waits until it has received responses from both scheduling nodes before proceeding. Responses returned may be marked as successful, i.e., the scheduler believes it can service the request within the client's requested deadline, or unsuccessful. Responses

also include an indication of how long the scheduler believes it should take to service the request. Using this information, the client can dispatch its request to the scheduler that provides a lower time estimate of the two tickets (representing the load of the two schedulers). As previously mentioned, this strategy is very effective in practice [49].

If neither ticket returns succesfully, then it is unlikely that the client will be able to service its request within its necessary deadline, and the client decides how next to proceed (performing other, less intensive work, re-issuing the request with a relaxed deadline, etc.).

In case a client is not not particularly concerned with the deadline of a given request (if, for example, it has no currency requirements), and additionally because it may be necessary for a particular request to never be rejected from percolating to the data store, MicroFuge also allows requests to be marked as best-effort.

Similar to requests with hard deadlines, best-effort requests wait on responses from both servers before deciding which node to use for scheduling. Both succesful and unsuccesful tickets are returned to the client with the scheduler's estimated completion times, and the client chooses the server with the best of the two provided times.

Unlike requests with hard deadlines, however, when servicing requests that are about to miss their best-effort deadlines (or have already), the scheduler has the option to re-queue the best effort request with a new, extended deadline. In this manner, best-effort requests that will miss their deadlines can be further delayed to the benefit of other requests (both best-effort and hard-deadline) in the system if necessary.

Write requests in MicroFuge are handled identically to read requests, but are always treated as best-effort to ensure that updates are consistently pushed through to the data store.

Having blocked on a ticket, the client waits until it is contacted by the scheduler to proceed. It then contacts the data store and performs its operation. Once the request has completed at the data store, the client contacts the cache, providing it with the request's value (so the cache layer can determine whether or not to retain the value). The client also concurrently sends a message to the scheduler that the ticket has been satisfied.

5.2 Sample Request

A timeline demonstrating the scheduling journey of a sample read request in the MicroFuge system is provided in Figrue 5.1. The read request shown in the figure is not a best-effort

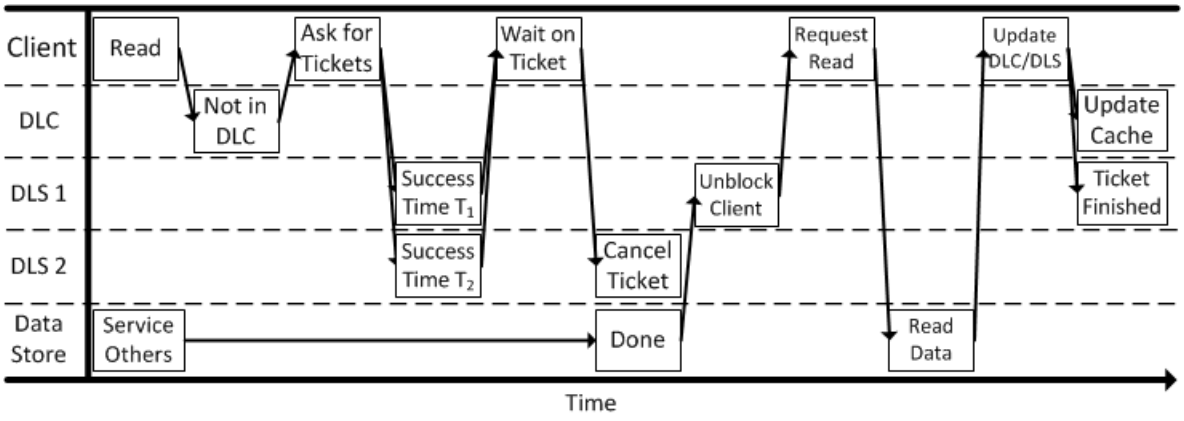


Figure 5.1: Sample timeline for a read request from a client. For this request, the requested item is not contained in the cache and both schedulers accept the ensuing ticket requests.

request. In this section, we give a breakdown of the stages of the protocol that the request follows.

Initially, we assume that the data store is busy processing another request. The client then issues its read request to one of the DLC nodes. The cache node determines that the request item is not resident (represented by 'Not in DLC'), and informs the client that it must contact the scheduling layer (represented by 'Ask for Tickets').

The client determines two scheduling nodes, each responsible for a replica of the data that it wishes to retrieve, and simultaneously makes ticket requests to both of them, including its deadline requirements with the request.

Scheduler one (DLS 1) informs the client that it can service the request within a given amount of time, as does the second scheduler (DLS 2). This is shown in Figure 5.1 as 'Success Time T1' and 'Success Time T2', respectively. The client compares estimated completion times from both the schedulers, and determines that scheduler one (DLS 1) is likely to complete the request sooner. The client sends the second scheduler a cancel notification (represented by 'Cancel Ticket'), and waits on a response from the first scheduler (represented by 'Wait on Ticket').

Eventually, the data store finishes its previous operation, and the client that had requested the data (not pictured in the figure, for simplicity) informs the scheduler that its ticketed operation is complete. Scheduler one then contacts the sample client, which proceeds to make its request directly to the data store (represented as 'Read Data').

Once the data store has finished processing the client's read request, it responds to the client with the requested value. The client in turn concurrently contacts the cache (to inform it of the request's value), and the scheduler (to inform it that the ticketed request has completed). The cache makes any necessary updates to the cached data set (represented as 'Update Cache'), and the scheduler updates its latency meta data. Having completed all necessary steps, the protocol is complete.

5.3 Fault Tolerance

For MicroFuge, we do not provide any explicit fault-tolerance mechanism as both the components of the system (DLC and DLS) maintain only a soft-state of the system. Also, having a replicated component would increase the response time of the middle layer, as now the processes would need to synchronize the states among all the replicas. In a latency-critical system, any increase in response-time of a middle layer is not desirable. Therefore, we decided to have a failure model which will ignore any failure of these middle layers and continue the control-flow by using a timeout mechanism.

In the event of failure of any of DLC or DLS, MicroFuge's client library behaves as if the component was not there and transparently handles the failure. For example, if a DLS node fails, the MicroFuge client will wait for the timeout (provided by Apache Thrift framework [1]) from the node after which it will make the request directly to the corresponding data server of the back-end cloud storage system.

This is not to say that fault-tolerance is not important. However, the performance would depend on the failure recovery architecture of the component. If we are able to bound the latency introduced by the failure recovery mechanism, it may be possible to have fault tolerance with minimal loss of performance. However, any such scheme would need to be evaluated independently and can definitely be considered as a future extension to MicroFuge.

Chapter 6

Experimental Evaluation

The performance characteristics of a cloud storage system are critical to the success of the system in a multi-tenant environment, especially when employed for latency-sensitive applications. In this chapter, we present an evaluation of the MicroFuge system and examine its effectiveness in providing performance isolation for cloud storage systems. We begin by a description our experimental setup and the test application we use to benchmark performance of the system. Next we evaluate the contribution of our caching layer (Deadline Cache) and show how deadline-aware caching policies can improve the performance of a cloud storage environment without any support from it. Lastly, we demonstrate the effectiveness of our scheduling layer (Deadline Scheduler) in restricting the percentage of requests that miss their SLOs by using its deadline-aware admission control.

6.1 Experimental Setup

For performance characterization purpose, an eight-node test cluster was configured, as shown in Figure 6.1. The first 4 machines were configured as servers, each running our simple cloud storage system (detailed in Section 6.1.1) and MicroFuge services. The other 4 machines were used as clients, each running an instance of a modified version of the Yahoo! Cloud Serving Benchmark (or YCSB) framework [23] (see Section 6.1.2), a standard benchmarking tool for key-value pair-based storage systems. All the machines are in the same internal network, simulating the network setup of a datacenter.

Each server machine is a Quad-Core AMD FX(tm)-4100 server running at 3600 MHz with 2GB of main memory. These nodes are configured with two Seagate Barracuda 500

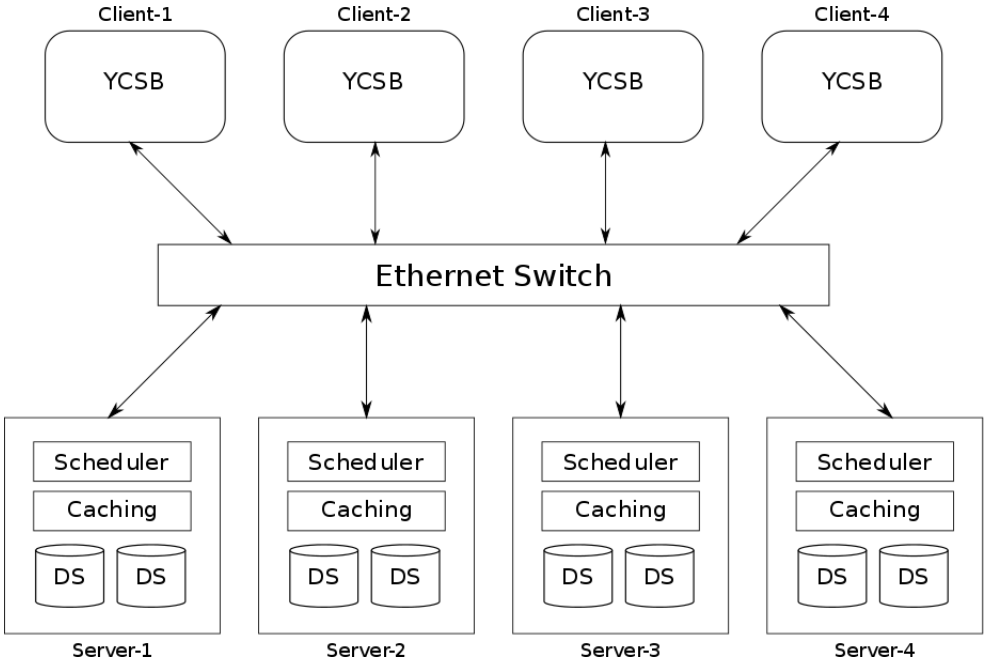


Figure 6.1: 8-node test-Cluster setup with 4 servers and 4 clients.

GB SATA 3.0 Gb/s 7200 RPM Hard Drives and two Intel PRO/1000 Gigabit Network Adapters, so that each machine can run two instances of our simple cloud data-store without any interference at the disk or network layer. For evaluating the performance of our system, we use up to 4 dual-core AMD Athlon(tm)-7750 machines running at 2700 MHz with 4GB of RAM. They are configured with one Western Digital 500GB SATA 3.0Gb/s 7200 RPM HDD and one Intel PRO/1000 Gigabit Network Adapter each. All machines were configured to run 64-bit Ubuntu-11.10 server edition on top of 3.0.0-26 of the Linux kernel. All disks use the recommended ext-4 filesystem, with Ubuntu server edition's default settings.

Both the components of MicroFuge, together with our DataServers use Apache Thrift [1], for marshalling, unmarshalling and communication over the network. Thrift is an an open source cross-language Remote Procedure Call (RPC) framework, originally developed at Facebook. In our current setup, we use Thrift version 0.9.0.

As discussed earlier in Chapter 1, MicroFuge can be deployed on top of any cloud storage system. However, to keep things simple, we have implemented our own distributed data storage layer called DataServer for the purpose of evaluating MicroFuge (see Section

6.1.1). To characterize the performance of different configurations of the MicroFuge system running on top of our persistent storage layer, we use a variety of workloads provided by YCSB. The remainder of this section discusses these two systems in detail.

At the software level, our cluster has following service configurations:

- **DataServer Servers**
  As shown in Figure 6.1, there are a total of eight DataNode Servers (DS) running in the cluster, with each server machine hosting two instances of DS on two different disks. We are able to multiplex these processes in this manner because they are not CPU or memory intensive but are I/O bound. Therefore, a process running on its own independent disk is a sufficient simulation of a process running on an independent machine.

- **Caching Servers**
  In a production setup, caching server processes run on dedicated machines. However, for the purpose of basic evaluation of the system, the caching layer (either Memcached or Deadline Cache) is run on the same set of server machines as other services in our setup. Each server machine is running an instance of q cache server with 1.5GB of cache-space, representing a total cache size of 6GB.

  Note that in our setup, most of the main memory of the servers is used by the external cache, leaving only the minimum required memory for the FileSystem cache. Specifically, each server machine has 2GB of RAM, out of which 1.5GB is occupied by the caching layer, around 200MB-350MB is taken by the OS and the remaining 200-250MB is used by the FileSystem cache. Using such a configuration, we show that it is more beneficial towards providing performance isolation in a persistent storage system to dedicate memory to an external distributed caching layer than it is to dedicate that same memory to the FileSystem cache.

- **Scheduler Servers**
  In a production setup, where we have a bigger deployment of a cloud storage system, we have dedicated machines running our scheduler service. As discussed in Chapter 4, each instance of DLS can handle multiple data-servers. These machines running DLS should have strong processing power, as each storage operation requests two scheduler processes for scheduling a ticket to corresponding data-servers (outlines in detail in Chapter 5). However, for the purpose of these experiments the scheduler servers share machines with the caching and persistent storage services.

- **Benchmarking System**
  An instance of our modified version of YCSB (details in Section 6.1.2) runs on each

of our four client machines. Every YCSB process is capable of running multiple client threads, each of which issue a sequence of I/O operations to our storage system.

6.1.1 DataServer System

We use our own cloud storage system, called DataServer, to avoid the need of performance-tuning of more complex storage systems like HBase [5] or MongoDB [8]. DataServer is a simple distributed key-value pair-based storage system, using leveldb [6] as its embedded key-value storage library. DataServer tunes several of leveldb's parameters to restrict the extent of caching and computing resources used by the server process. This makes it easy to understand the effect of the external caching layer and request ordering. All the major parameters used in the current version of DataServer are listed in Table-6.1.

LevelDB Parameter	**Value**
Key Comparator	Default
Write Buffer Size	64MB (default : 4MB)
Block Size	2KB (default : 4KB)
Maximum Open Files	64 (default : 1024)
Block Cache	Default (8MB)
Bloom Filter	Not Used
Compression Type	Snappy Compression

Table 6.1: Parameters used for LevelDB

Drawing inspiration from Dynamo and Cassandra, we use a simple hash-based scheme to determine which keys map to which servers. By default, we use three-way replication: the servers are determined by the key hash plus the next two servers in the ID space.

DataServer provides a modest CRUD API, resembling a typical cloud storage system. It has a very simple data storage model inspired by Google File System [30]. Although it supports the concept of a user-level data partition (generally referred to as a tablet in other cloud storage systems), for our current set of experiments we employ one big tablet which contains all the data. Like other cloud storage systems, the basic unit of data distribution in DataServer is a tablet. DS also supports dynamic replication of tablets based upon the usage pattern of the data.

6.1.2 Benchmarking System

To evaluate the performance of our deployment, we use an industry-standard benchmarking tool called Yahoo! Cloud Serving Benchmark (YCSB) [23]. YCSB is designed for benchmarking key-value pair-based cloud storage systems and provides a variety of workloads representing different classes of real-world workloads. Since typical key-value pair-based cloud storage systems do not have the notion of response time SLOs, YCSB does not provide any means for specifying these SLO values in its workloads. As MicroFuge incorporates deadline information for resource allocation and scheduling, we modified YCSB to provide deadline metadata to the generated storage requests.

Range	Deadlines	Proportion
Range-1	10-30 ms	0.2
Range-2	30-100 ms	0.3
Range-3	100-1000 ms	0.5

Table 6.2: Distribution of deadlines in the Workload

The assignment of deadlines to different requests is based on the hash of a request's key. Since these deadlines are not randomly generated, a particular key will be assigned the same deadline value every time. This simulates a practical scenario, where a given key belongs to one particular request-type, which in turn is likely to have the same response time SLO each time it needs to be serviced by the storage system. For the numbers presented in this section, YCSB produces response time deadlines in the range of 10 to 1000 ms. In this workload, 20% of requests have deadlines between 10 to 30 ms, 30% have deadlines between 30 to 100 ms, and the remainder have deadlines ranging from 100 to 1000 ms, as shown in Table-6.2. Within these ranges, deadlines follow a uniform distribution. This is representative of a real-world application, where a limited variety of request types (in this case three) are sent to the back-end storage system. This has been discussed with an example in Section 3.1.1. However, the exact distribution used for these set of experiments is synthetic. In a real-world deployment the distribution of deadlines for requests may vary.

Unless otherwise specified, the data-set used for these experiments has 20 million records and is around 21.6GB in size. Our distributed cache system (either Memcached or DLC) has an overall capacity of 6GB, which is around $1/3rd$ of the total data-set size, representing a realistic deployment scenario.

For numbers presented in the following sections, we used a standard YCSB setting of 13-byte keys and 1KB value size to generate the data-set. For transaction workloads,

we used predefined YCSB *coreworkloads*, which range from insert-heavy to read-heavy to a mixture of both types of operations. Each run of these experiments performs 100K operations,with a a Zipfian distribution of requests. All the experiments are reported with an average of five independent runs, computed with 95% confidence interval.

In the next section, we show some of the results of our comprehensive evaluation of the MicroFuge system. We start by evaluating the performance of our caching system and show that it is comparable to basic performance characteristics of Memcached. We then show how deadline-aware caching policy leans towards evicting data items with longer deadlines as compared to the ones with shorter deadlines. Going further, we illustrate how Deadline Cache is effective in providing performance isolation to underlying cloud storage systems by meeting more SLOs. Finally, we show how Deadline Scheduler plays a principal role in restricting the percentage of requests missing their deadlines, using its deadline-aware admission control policy.

6.2 Performance of Deadline Cache

This section shows the performance characteristics of our caching system compared to those of Memcached. The purpose of these experiments is to show that the two systems are comparable in terms of space-efficiency and throughput at different levels of load. It is necessary to prove that the comparison between the two systems is fair, and that it is the deadline-aware caching policy of Deadline Cache (and not its engineering effort), that helps it to meet more SLOs compared to Memcached.

It is important to recognize that our current implementation of MicroFuge is just a prototype and not much optimization effort has gone in to making it more efficient as a cache. On the other hand, Memcached is a highly optimized caching system which is widely used in industry settings. Therefore, when it comes to comparison of basic performance characteristics like throughput and space-efficiency of Deadline Cache and Memcached, the latter may outperform the former.

6.2.1 Cache Hit-Rate

Figure 6.2 shows the overall cache hit-rate avergaed over 5 runs of the two caching systems for 96 concurrent clients with a 95% confidence interval. Memcached achieves 2 to 3% higher cache hits on average when compared to Deadline Cache. The principal reason for such behaviour is that Memcached has a strict LRU caching policy, while the cache policy

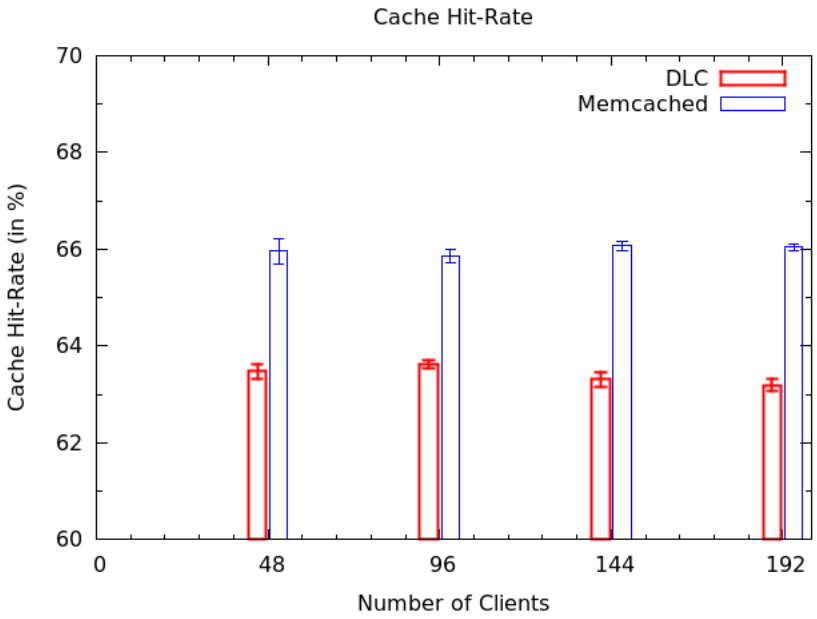


Figure 6.2: Overall cache hit-rate at different levels of load

of Deadline Cache is similar to LRU but with additional deadline-oriented considerations (see Section 3.1.2). Therefore, the chance of finding an item in the cache when it has recently been used, is greater for Memcached than for Deadline Cache. This is because the eviction policy of Deadline Cache not only considers the recency of data items but also incorporates their associated deadline metadata.

Cache System	Cache Space
Deadline Cache	3004MB
Memcahced	2844MB

Table 6.3: Cache-space taken by each system in storing 200K regular YCSB records.

In conjunction with the difference between the eviction policies of the two systems, the fact that Deadline Cache has not been fully optimized to efficiently use memory space results in DLC storing less data items than Memcached in the same amount of memory-space. This also contributes to lower-than-average cache hit-rate for Deadline Cache when compared against the hit-rates of Memcached. As shown in Table-6.3, Memcached is more

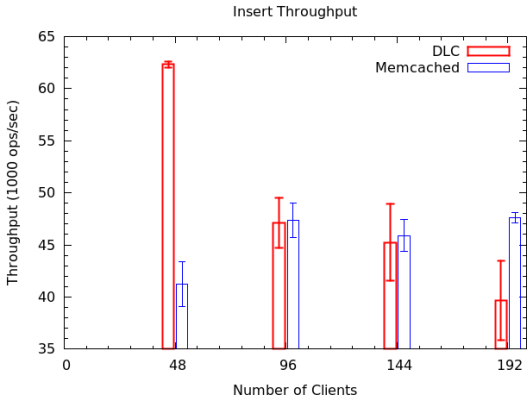
space-efficient than Deadline Cache. At least part of the reason that Deadline Cache storage occupies more space can be attributed to the fact that it stores additional deadline metadata with each data item, in contrast with Memcached which only stores the data items themselves.

6.2.2 Cache Throughput

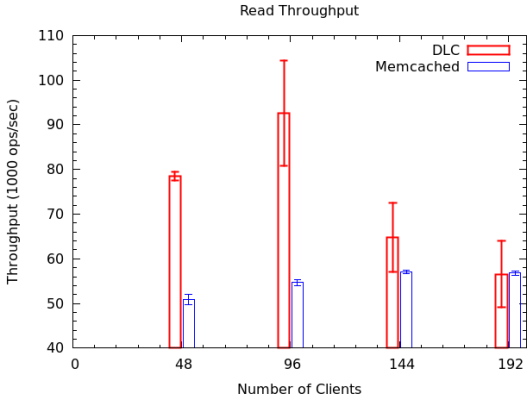
Figure 6.3 shows the throughput of the two systems under different YCSB workloads. For these experiments, each workload performed 200K operations over 200K records. The request distribution of the workloads was set to the default zipfian distribution.

Figure 6.3 (a) shows the throughput of INSERT operations for different numbers of clients. The cache size for this experiment was big enough to avoid eviction, ensuring that insert performance was not affected by eviction procedure of the systems. As shown in the figure, the performance of Deadline Cache looks better for a small numbers of clients. As the number of clients in the system increases, the throughput of DLC decreases accordingly. Memcached, on the other hand, is seemingly able to retain a steady rate of 47K ops/sec even with an increased number of clients. The better scalability of Memcached can be attributed to additional complexity introduced by the multi-queue architecture of Deadline Cache. Furthermore (and as previously mentioned) Memcached has seen significant amounts of optimization over its years of development.

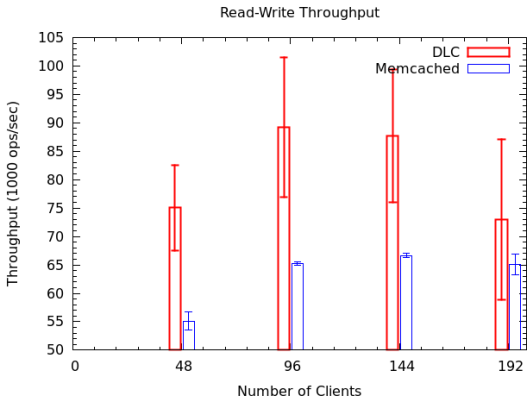
Figure 6.3 (b) & (c) shows the throughput of READ/UPDATE operations for workload C and B respectively. Workload-C represents a read-only workload, while Workload-B represents a mixed workload with 95% read operations and 5% update operations to existing keys. Deadline Cache clearly outperforms Memcached by a considerable margin. The predominant reason for the low throughput of Memcached (compared against that of DLC) is fine-grained cache locking mechanism used in the latter. As discussed in [39], the global cache lock in Memcached becomes a performance bottleneck for more than 4 threads. As we see in Figure 6.3 (b) & (c), the throughput of DLC peaks at around 96 clients, before showing a decrement of more than 10% as clients are increased to 196. Such performance behaviour can be attributed to the fact that even DLC's fine grained locking is not sufficient to handle this high level of load. In such scenarios, cache locks start to act as a performance bottleneck, similar to the bottlenecks that Memached experiences even at lower levels of load with few clients.



(a) INSERT performance



(b) READ performance in Workload-C (100% reads)



(c) READ/UPDATE performance in Workload-B (95% reads, 5% updates)

Figure 6.3: Average throughput of two caching systems at different load levels for a variety of real-world workloads provided by the Yahoo! Cloud Serving Benchmark.

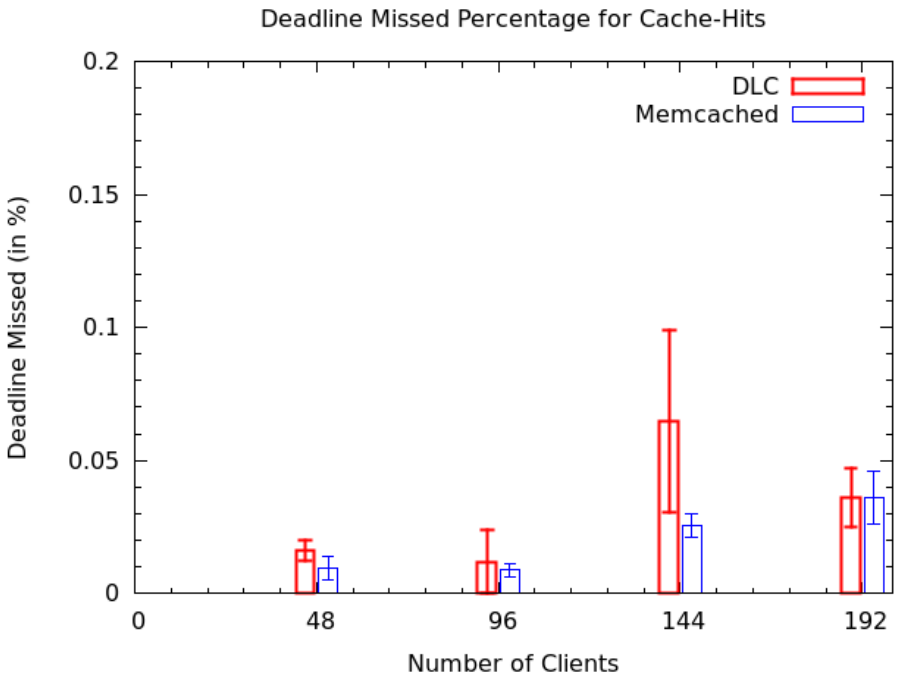


Figure 6.4: Deadline missed percentage of requests which were a cache hit

6.2.3 Cache Response Time

To keep the design of MicroFuge simple throughout its design and development process, we followed the assumption that every request which is a cache hit should also meet its associated deadline. Figure 6.4 shows that the percentage of requests which missed their deadlines, even when the item was found in the cache, is below 0.1% across all the load levels. This validates our assumptions that only an insignificant fraction of such requests miss their deadlines.

Together with the results presented in the previous sub section, this experiment shows that neither system is faster or more efficient than the other at servicing cached requests. Therefore, the throughput or the response time of the caching system cannot be identified as a cause for SLO misses in cloud storage systems.

It is to be noted that in this set of experiments the deadline-aware nature of Deadline Cache does not play any role. Rather, the effects of DLC's deadline-conscious design are visible when we show the DataServer's ability to meet SLOs given the underlying

interactions of both systems, which is the main focus of this thesis. In the next section, we demonstrate how each of these systems helps the cloud storage system (DataServer) reduce the fraction of requests that miss their response time SLOs.

6.3 Deadline Aware Caching

This section aims to show how the deadline-aware caching policy of Deadline Cache helps reduce the number of requests with short-lived deadlines that miss their SLO objectives. This is achieved by the system through intelligent caching decisions that increase cache hit-rates for short deadline items. Since these requests are more likely to miss their deadlines in the event of fetching the data from the disk, Deadline Cache is designed to keep these items in the queue for a greater amount of time than items with longer deadlines. The strategy is further supplemented by the fact that objects with longer deadlines are relatively more likely than objects with shorter deadlines to meet their goals if they are not served from the cache, implying that they may be removed at a higher frequency without negatively affecting system performance.

Figure 6.5 shows a detailed graph of cache hit-rates for requests with varying deadlines (see Section 6.1.2). These ranges are marked with horizontal violet lines in the figure. From the figure, it is clear that the aggregate cache hit-rate (represented by thick horizontal lines) of Memcached (66.06%) is higher than that of Deadline Cache (63.57%). As discussed in Section 6.2, besides space-efficiency, the primary reason for such behaviour is the difference in the eviction policies between the two systems. Memcached's eviction policy does not differentiate between the items on the basis of their associated deadlines. On the other hand, Deadline Cache's eviction policy considers the SLO requirements and stores more items with shorter deadlines, evicting objects with longer deadlines in their favour if necessary.

Owing to the special treatment of shorter deadline items by Deadline Cache, the cache hit-rate for first two ranges of deadlines is higher for Deadline Cache than it is for Memcached. This is showcased in Figure 6.6, which shows the cache hit rate for the first two ranges of deadlines, for the same experiment as in Figure 6.5. Since, items belonging to the shorter range have a major share of the total requests that miss their deadlines, it is better to aim for a higher cache hit rate for them.

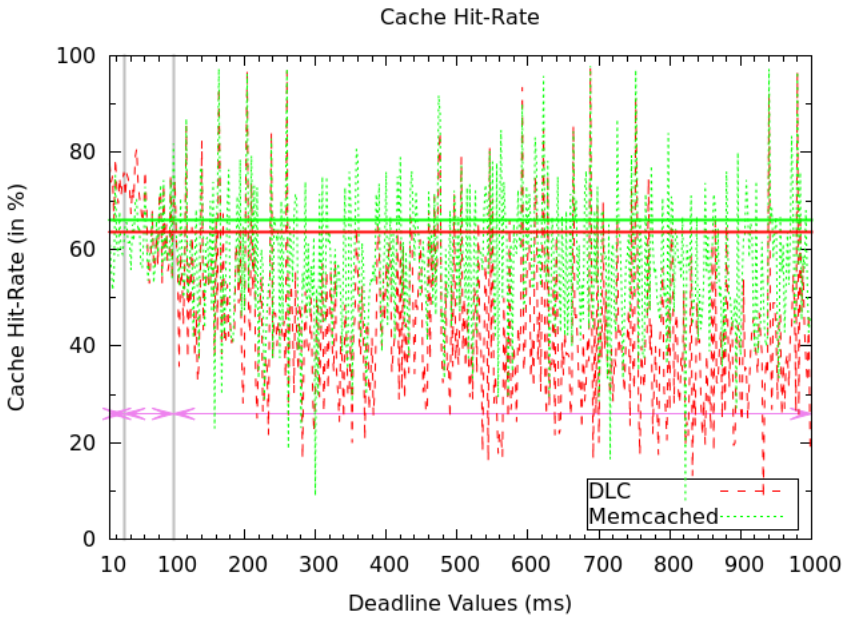


Figure 6.5: Cache hit-rate versus deadline values at a moderate load of 96 concurrent clients

6.4 Contribution of Caching in meeting SLOs

As shown in the previous section (Section 6.3) the cache hit-rate of Deadline Cache is higher than that of Memcached for items with shorter deadlines. However, overall cache hit-rates are higher for Memcached. This compromise in design of our system pays off when we try to reduce the fraction of requests that miss their deadlines.

Figure 6.7 shows the deadline miss percentage for all different ranges of deadlines. As expected, due to the high cache hit-rate for the first two ranges in Deadline Cache the deadline miss percentage in them is less for DLC than it is for Memcached. However, Memcached outperforms Deadline Cache in the third range. Also, as shown in Figure 6.7 (d)Deadline Cache misses less deadlines overall (14.40%) than Memached does (16.88%). It can therefore be concluded that Deadline Cache is more effective in reducing the deadline misses in a cloud storage system workload.

The above results are for lightly loaded systems, running 48 concurrent clients. To show that our approach is similarly effective for high load levels in the system, we performed

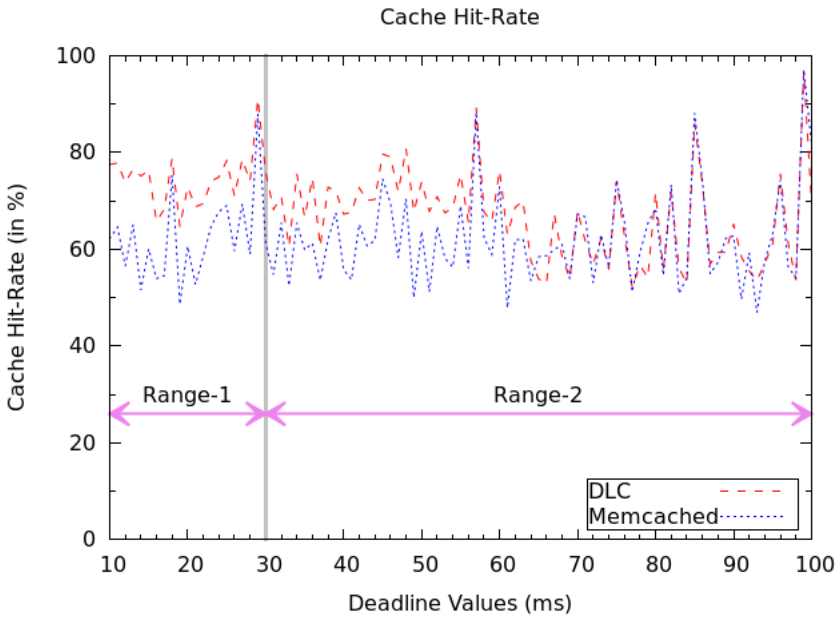
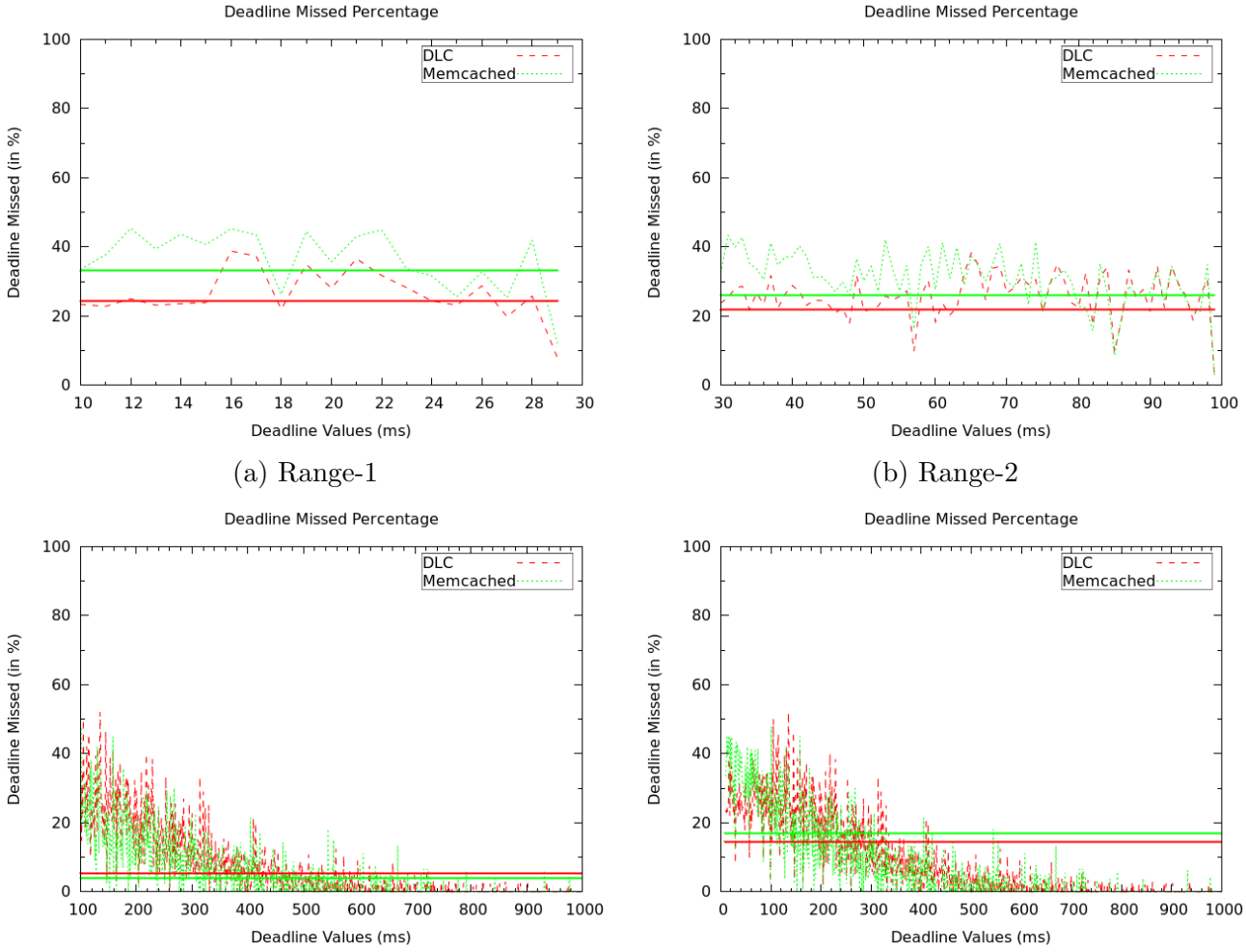


Figure 6.6: Cache hit-rate for first 2 ranges of deadlines for different deadline values at a moderate load of 96 concurrent clients

the same set of experiments with 192 concurrent clients. As shown in Figure 6.9, in a setting with high load, though the difference between the performance of the two systems decreases, the overall trend remain the same. The reason for such a behaviour, as discussed in the previous sub-section (Section 6.2.2), is that at higher loads Deadline Cache starts to have the same performance bottlenecks as Memcached. In summary, Memcached is good for maximizing the cache-hit rate but since it is not deadline-aware it less effective in helping the underlying cloud storage system to miss less deadlines for a wide range of system load.

Finally, we compare the deadline meeting capability of our cloud storage system deployment in different cache configurations. Figure 6.8 gives a summary of the performance of three systems in terms of meeting SLOs. As expected, the DataServer without any caching (represented by the green bar) performs worst among all three configurations, while Deadline Cache performs the best across different degrees of system load.



(a) Range-1

(b) Range-2

(c) Range-3

(d) All Ranges

Figure 6.7: Deadline Miss Percentage for data items in different ranges are shown in (a), (b) and (c) at a load of 48 concurrent clients. (d) shows the same quantity but includes all the deadlines.

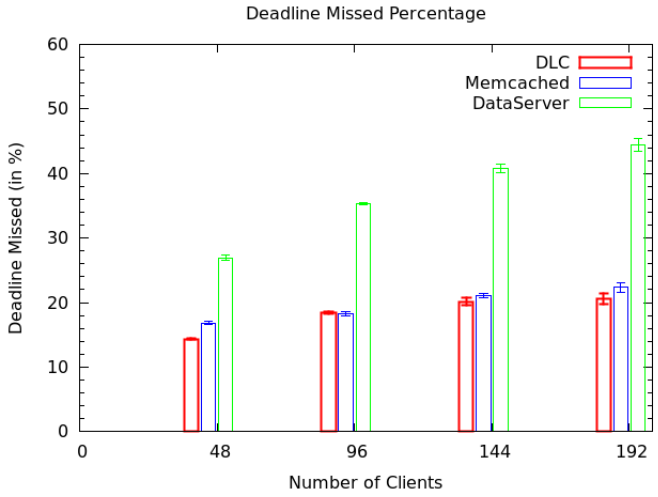
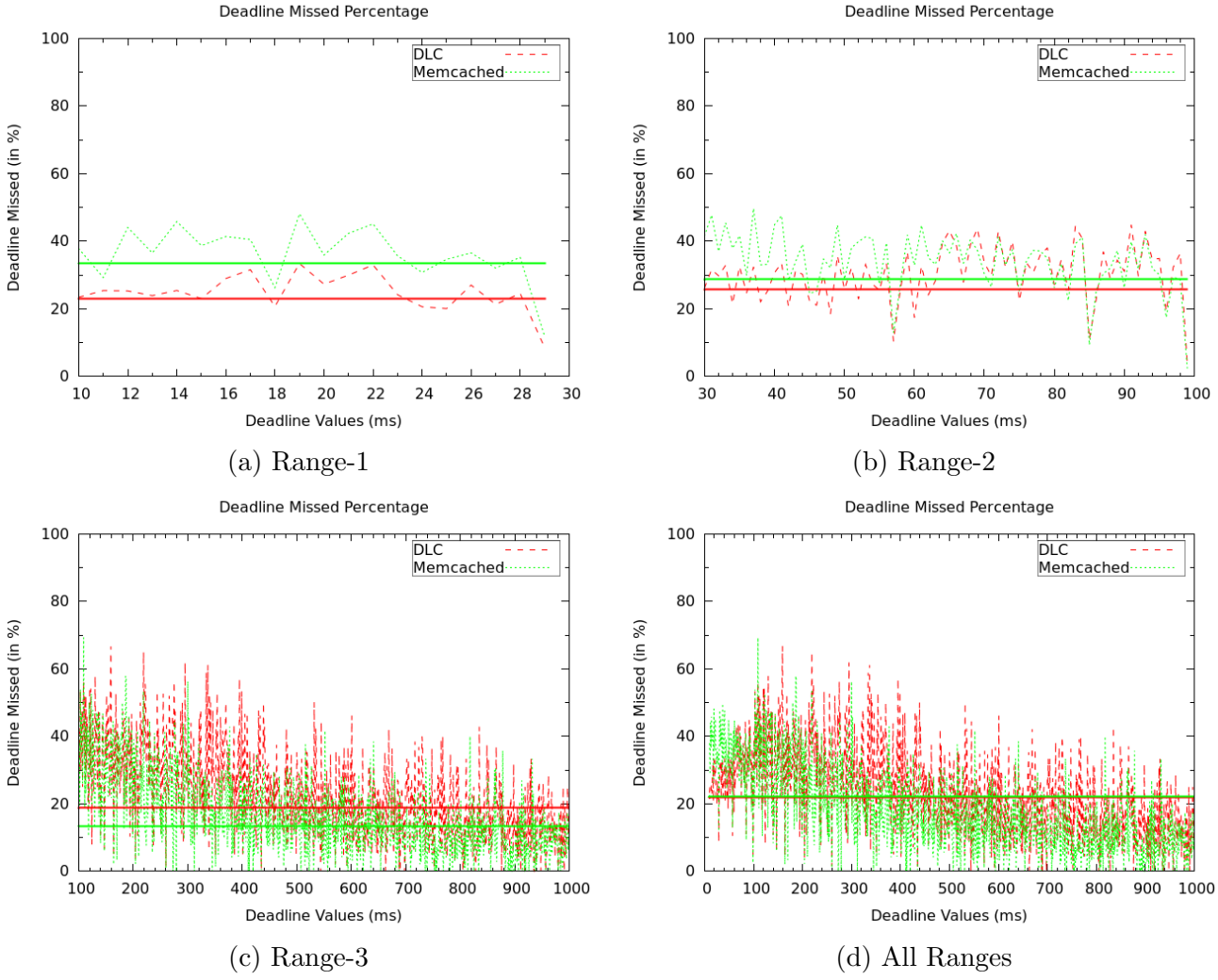


Figure 6.8: Deadline Missed Percentage (including all 3 ranges)

6.5 Contribution of Scheduling in meeting SLOs

Section 6.4 discussed the contribution of the caching layer in our overall objective of missing less SLOs. We have already established that deadline meeting performance of DLC outruns Memcached. In this section, we will demonstrate the improvements gained by the introduction of our scheduling layer (DLS). This layer can be deployed on top of any cloud storage system and external caching layer. Since Deadline Scheduler and Deadline Cache are not coupled in anyway and we want to focus on contribution of the scheduler, we will be using DLC to represent any caching layer.

We first show how our deadline-aware scheduling helps to balance the load among the data-server nodes in the back-end cloud storage system, as an effect of which it reduces the number of SLOs being missed. All the requests in these set of experiments are best-effort (see Chapter 3) and any reduction in percentage of SLOs being missed can be attributed to our scheduling policies in DLS. Figure 6.10 gives a summary of performance of different systems in terms of meeting SLOs. It compares the performance of following systems : only DataServer layer with no caching (represented by 'DataServer'), DataServer layer with DLC (represented by 'DLC') and DLS over DataServer layer with DLC (represented by 'DLC+DLS'). Addition of scheduling layer seems to give an improvement of around 6-7% at high load. DLS does not show much improvement for lower levels of load as there is not much opportunity to schedule requests. As expected, putting scheduling on top of



(a) Range-1

(b) Range-2

(c) Range-3

(d) All Ranges

Figure 6.9: Deadline Miss Percentage for data items in different ranges are shown in (a), (b) and (c) at a high load of 192 concurrent clients. (d) shows the same quantity but includes all the deadlines.

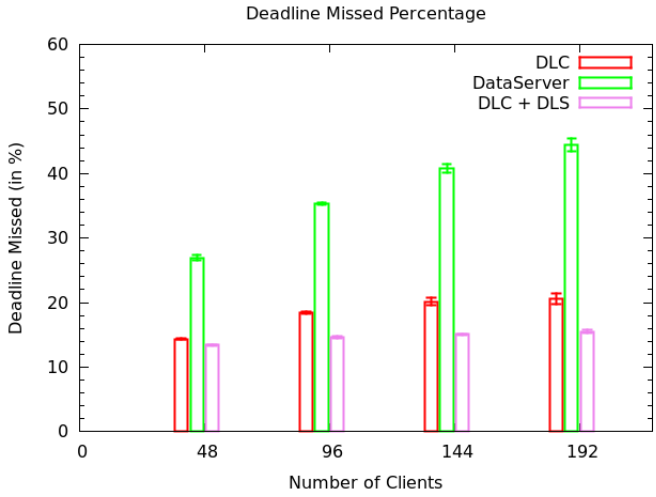


Figure 6.10: Contribution of Deadline Scheduler in reducing the deadline missed percentage for different levels of load. These numbers include the results for all 3 ranges of deadline.

our caching layer successfully reduce the number of requests missing their SLOs.

In the next set of experiments, we illustrate the effectiveness of our admission control mechanism in achieving performance isolation in cloud storage systems. Figure 6.11 outlines the contribution of admission control in containing the fraction of requests that miss their deadlines. The figure represents the same quantities as shown in figure 6.10 with the addition of a configuration DLS with admission control represented by 'DLC + DS (AC)'. As can be seen, with admission control turned on, Deadline Scheduler is able to restrict the SLO missed percentage around 10%, which was our target for these set of experiments.

As discussed in Chapter 4, to limit the number of requests that miss their deadlines we need to reject some requests. Figure 6.12 also includes the request rejection percentage represented as 'DLC + DS(AC) + Rejection'. As shown, we reject a limited fraction of total requests to restrict the fraction of requests missing their deadline, below our client's Service Level Agreement (or SLA).

To reject the requests which are not likely to meet their deadline, a naïve approach can end up rejecting majority of the short deadline requests. This will lead to starvation of these urgent requests (see Chapter 4). In our design, we selectively choose the requests which should be rejected to meet the tenant's SLA. Figure 6.13 shows that we reject the requests with deadlines from a wide range of values, and is not limited to short deadline

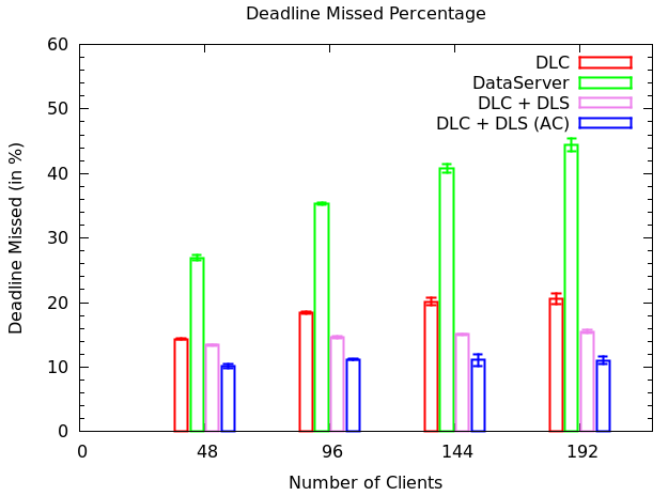


Figure 6.11: Effectiveness of Deadline Scheduler's admission control mechanism in restricting the percentage of requests that miss their deadlines. These numbers include the results for all 3 ranges of deadline.

requests. Having said that, since requests with shorter deadlines are the ones which are harder to meet their SLOs, a good portion of rejected requests comes from shorter range of deadlines.

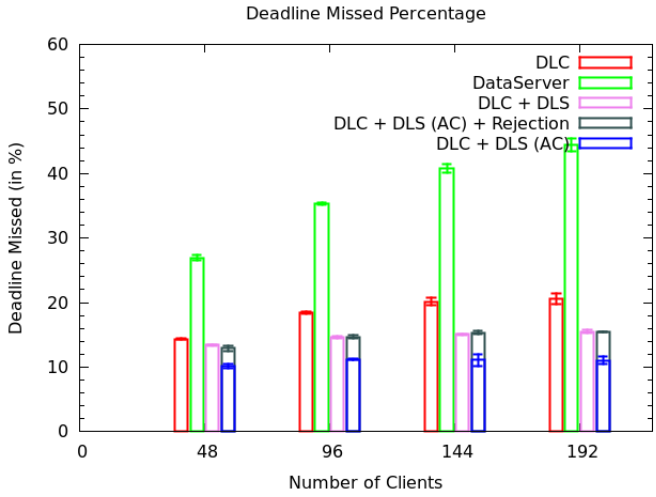


Figure 6.12: Rejection rate of requests by DLS's admission control mechanism versus load.

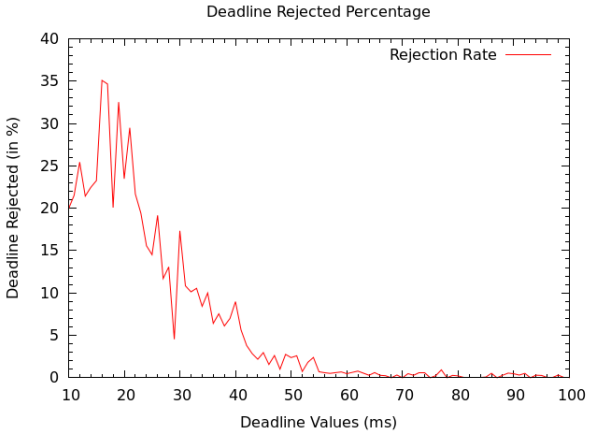


Figure 6.13: Rejection rate of requests by DLS's admission control mechanism versus deadline values for 192 concurrent clients.

Chapter 7

Conclusion

In this thesis we introduced MicroFuge, a new middleware application that provides distributed scheduling and caching services to cloud storage systems. MicroFuge focuses on deadline-awareness across all of its layers to help provide performance isolation that is typically difficult to accomplish inside multi-tenant systems. MicroFuge is built with the same API as Memcached, and is easy to layer over top of almost any cloud storage system.

MicroFuge's distributed cache layer, Deadline Cache, uses a tenant and deadline-aware cache eviction policy to isolate tenants and ensure that requests with short deadlines - which are more likely to be violated on a cache miss - are also more likely to be retained in the cache. This is performed with the use of multiple LRU queues based on deadline ranges.

Our system's distributed scheduling layer, Deadline Scheduler, uses a ticket-based scheduling system that helps to not only load balance among different data-servers, but service requests according to their deadline requirements. In addition to this, the scheduling layer collects latency metadata from completed requests, and uses this data to generate latency estimates for future requests. These estimates are, in turn, used to provide admission control, rejecting requests with deadlines that are unlikely to be met (based on the underlying performance model).

Through experimentation we have demonstrated that MicroFuge offers significantly better performance isolation than the current industry standard, Memcached.

References

[1] Apache thrift. http://wiki.apache.org/thrift/, 2013.

[2] Apache traffic server. http://trafficserver.apache.org, 2013.

[3] Ehcache: Java's most widely-used cache. http://www.ehcache.org/, 2013.

[4] Foundationdb. http://foundationdb.com/, 2013.

[5] Hbase. http://hbase.apache.org, 2013.

[6] Leveldb. http://code.google.com/p/leveldb/, 2013.

[7] Memcached. http://memcached.org, 2013.

[8] Mongodb. http://www.mongodb.org/, 2013.

[9] Project Voldemort. http://project-voldemort.com, 2013.

[10] Squid: Optimising web delivery. http://www.squid-cache.org/, 2013.

[11] Varnish cache. https://www.varnish-cache.org/, 2013.

[12] Robert Kilburn Abbott and Hector Garcia-Molina. Scheduling I/O requests with deadlines: A performance evaluation. In *Proceedings of the Real-Time Systes Symposium*. IEEE, 1990.

[13] Alfred V Aho, Peter J Denning, and Jeffrey D Ullman. Principles of optimal page replacement. *Journal of the ACM (JACM)*, 18(1):80–93, 1971.

[14] Chris Aniszczyk. Caching with twemcache. http://engineering.twitter.com/2012/07/caching-with-twemcache.html, July 2012.

[15] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[17] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[18] Ramon Caceres, Fred Douglis, Anja Feldmann, Gideon Glass, and Michael Rabinovich. Web proxy caching: The devil is in the details. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):11–15, 1998.

[19] David D Chambliss, Guillermo A Alvarez, Prashant Pandey, Divyesh Jadav, Jian Xu, Ram Menon, and Tzongyu P Lee. Performance virtualization for large-scale storage systems. In *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, pages 109–118. IEEE, 2003.

[20] Kavitha Chandra. Statistical multiplexing. *Encyclopedia of Telecommunications*, 2003.

[21] Marek Chrobak and John Noga. Lru is better than fifo. In *In Proc. 9th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 78–81, 1998.

[22] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[23] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 2010.

[24] Kyle Cordes. Youtube scalability talk. http://kylecordes.com/2007/youtube-scalability, July 2007.

[25] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, volume 7, pages 205–220, 2007.

[26] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. *Proc. 10th USENIX NSDI*, 2013.

[27] Bin Fan, Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 23. ACM, 2011.

[28] Brady Forrest. Bing and google agree: Slow pages lose users. http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html, June 2009.

[29] Joshua D. Gagliardi, Timothy S. Munger, and Donald W. Ploesser. Content delivery network, 10 2012.

[30] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[31] Steve Gilheany. Ram is 100 thousand times faster than disk for database access. http://www.directionsmag.com/articles/ram-is-100-thousand-times-faster-than-disk-for-database-access/123964, January 2003.

[32] Adam Wolfe Gordon and Paul Lu. Low-latency caching for cloud-based web applications. In *Proceedings of the 6th International Workshop on Networking Meets Databases*, 2011.

[33] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *USENIX Conference on File and Storage Technologies*, pages 85–98, 2009.

[34] Ajay Gulati, Chethan Kumar, Irfan Ahmad, and Karan Kumar. Basil: Automated IO load balancing across storage devices. In *USENIX FAST*, 2010.

[35] Ajay Gulati, Arif Merchant, and Peter J. Varman. pclock: An arrival curve based approach for QoS guarantees in shared storage systems. *ACM SIGMETRICS Performance Evaluation Review*, 35(1):13–24, 2007.

[36] James Hamilton. The cost of latency. http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx, October 2009.

[37] Martin Heller. Rest and crud: the impedance mismatch. http://www.infoworld.com/d/developer-world/rest-and-crud-impedance-mismatch-927, January 2007.

[38] VMware Infrastructure. Resource management with vmware drs. *VMware Whitepaper*, 2006.

[39] James T Langston Jr. Enhancing the scalability of memcached. http://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached-0, August 2012.

[40] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[41] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[42] Ron Kohavi, Randal M Henne, and Dan Sommerfield. Practical guide to controlled experiments on the web: listen to your customers not to the hippo. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 959–967. ACM, 2007.

[43] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *Operating Systems Review*, 44(2):35, 2010.

[44] Han Deok Lee, Young Jin Nam, Kyong Jo Jung, Seok Gan Jung, and Chanik Park. Regulating i/o performance of shared storage with a control theoretical approach. In *MSST*, pages 105–117. Citeseer, 2004.

[45] Xing Lin, Yun Mao, Feifei Li, and Robert Ricci. Towards fair sharing of block stoarge in a multi-tenant cloud. In *USENIX HotCloud '12*. ACM, 2012.

[46] Qiong Luo, Sailesh Krishnamurthy, C Mohan, Hamid Pirahesh, Honguk Woo, Bruce G Lindsay, and Jeffrey F Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 600–611. ACM, 2002.

[47] Joe McKendrick. Cloud computing's vendor lock-in problem: Why the industry is taking a step backward. http://www.forbes.com/sites/joemckendrick/2011/11/20/cloud-computings-\vendor-lock-in-problem-why-the-industry-is-taking-a-step-backwards/, November 2011.

[48] David T McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority mechanisms for oltp and transactional web applications. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 535–546. IEEE, 2004.

[49] Michael Mitzenmacher. How useful is old information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1):6–20, January 2000.

[50] Fiona Fui-Hoon Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163, 2004.

[51] Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. SQLVM: Performance isolation in multi-tenant relational databases-as-a-service. In *6th Biennial Conference on Innovative Data Systems Research*, January 2013.

[52] Microsoft Developer Network. About windows azure caching. http://msdn.microsoft.com/en-us/library/hh914161.aspx, October 2012.

[53] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. An optimality proof of the lru-k page replacement algorithm. *J. ACM*, 46(1):92–112, 1999.

[54] Paul Saab. Scaling memcached at facebook. http://www.facebook.com/note.php?note_id=39391378919, December 2008.

[55] Salvatore Sanfilippo and Pieter Noordhuis. Redis, 2010.

[56] Prashant J Shenoy and Harrick M Vin. Cello: a disk scheduling framework for next generation operating systems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 26, pages 44–55. ACM, 1998.

[57] David Shue, Michael J. Freedman, and Anees Shaikh. Fairness and isolation in multi-tenant storage as optimization decomposition. *ACM SIGOPS Operating Systems Review*, 47(1):16–21, 2013.

[58] Citrix Systems. What is caching? http://www.citrix.com/glossary/caching.html, 2013.

[59] Sandeep Uttamchandani, Li Yin, Guillermo A Alvarez, John Palmer, and Gul A Agha. Chameleon: A self-evolving, fully-adaptive resource arbitrator for storage systems. In *USENIX Annual Technical Conference, General Track*, pages 75–88, 2005.

[60] Matthew Wachs and Gregory R Ganger. Co-scheduling of disk head time in cluster-based storage. In *Reliable Distributed Systems, 2009. SRDS'09. 28th IEEE International Symposium on*, pages 278–287. IEEE, 2009.

[61] Matthew Wachs, Lianghong Xu, Arkady Kanevsky, Gregory R Ganger, et al. Exertion-based billing for cloud storage access. *Proc. HotCloud*, 2011.

[62] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Ion Stoica, and Randy Katz. Sweet storage SLOs with frosting. In *USENIX HotCloud '12*. ACM, 2012.

[63] Theodore M Wong, Richard A Golding, Caixue Lin, and Ralph A Becker-Szendy. Zygaria: Storage performance as a managed resource. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 125–134. IEEE, 2006.

[64] Helen Yang. 2 seconds as the new threshold of acceptability for ecommerce web page response times. http://www.akamai.com/2seconds, September 2009.