2014 IEEE 34th International Conference on Distributed Computing Systems

MicroFuge: A Middleware Approach to Providing Performance Isolation in Cloud Storage Systems

Akshay K. Singh, Xu Cui, Benjamin Cassell, Bernard Wong and Khuzaima Daudjee
Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
{*ak5singh, xcui, becassel, bernard, kdaudjee*}*@uwaterloo.ca*

***Abstract*—Most cloud providers improve resource utilization by having multiple tenants share the same resources. However, this comes at the cost of reduced isolation between tenants, which can lead to inconsistent and unpredictable performance. This performance variability is a significant impediment for tenants running services with strict latency deadlines. Providing predictable performance is particularly important for cloud storage systems. The storage system is the performance bottleneck for many cloud-based services and therefore often determines their overall performance characteristics.**

In this paper, we introduce MicroFuge, a new distributed caching and scheduling middleware that provides performance isolation for cloud storage systems. MicroFuge addresses the performance isolation problem by building an empirically-driven performance model of the underlying storage system based on measured data. Using this model, MicroFuge reduces deadline misses through adaptive deadline-aware cache eviction, scheduling and load-balancing policies. MicroFuge can also perform early rejection of requests that are unlikely to make their deadlines. Using workloads from the YCSB benchmark on an EC2 deployment, we show that adding MicroFuge to the storage stack substantially reduces the deadline miss rate of a distributed storage system compared to using a deadline oblivious distributed caching middleware such as Memcached.

***Keywords*-caching; middleware; performance isolation; storage; scheduling**

I. Introduction

Cloud computing has had a transformative effect on how businesses host their online services and manage their computational needs. As increasing numbers of users take advantage of the cloud, the associated rise in resource demands has cloud providers focusing on efficiently monitoring and allocating resources between users. By consolidating services from different tenants onto the same physical machines, a cloud provider can significantly increase resource utilization. This in turn allows cloud providers to offer a price-competitive hosting service to their tenants.

However, service consolidation can lead to poor performance if multiple tenants require the use of their resource reservations concurrently. For a resource such as memory, where tenants are generally more concerned about capacity than throughput and access latency is largely unaffected by concurrent access, resource sharing does not affect performance unless there is capacity oversubscription. Unfortunately, the opposite is true for storage where throughput and latency are of much greater concern than capacity, and resource sharing can significantly degrade a storage system's performance. This can lead to highly unpredictable performance for cloud storage clients that is unacceptable for those that have strict access latency requirements.

Therefore, it is critically important for cloud storage providers to offer performance isolation in their storage systems. Perfect isolation ensures that client performance is completely unaffected by other clients. This generally requires dedicated disks per client, significantly reducing resource utilization and inflating operating costs. Many clients instead prefer to enter into a performance-based Service-Level Agreement (SLA) with their cloud provider, in which they specify their performance requirements that must be met in spite of competing requests [1]. One of the key elements of an SLA is the response time service level objective (SLO) which can be naturally represented as request deadlines. Along with the response time SLO, an SLA ensures predictable performance for the clients and, compared to perfect isolation, provides additional resource sharing opportunities for the cloud storage providers.

In this paper, we introduce MicroFuge, a distributed middleware that tackles the performance isolation problem by building a lightweight and accurate performance model of the underlying storage system using measured data, and then adding a deadline-conscious, performance model-driven caching and scheduling layer to the cloud storage stack. The performance modeling component is crucial for determining cache eviction, scheduling, and load-balancing policies that minimize deadline misses for a given storage system. The MicroFuge middleware layer is similar to the external caching layer that is commonly used in most web service deployments.

Using this additional middleware abstraction, we demonstrate that cloud storage systems are able to effectively provide much stronger performance isolation for multiple cloud tenants. We define performance isolation as a property where a tenant can meet its performance requirements in spite of concurrent actions from other tenants, which closely

1063-6927/14 $31.00 © 2014 IEEE
DOI 10.1109/ICDCS.2014.58

models real-world, performance-based SLAs. By targeting the caching and scheduling layers, which only hold soft-state, and offering the same caching interface as Memcached, we greatly reduce MicroFuge's barrier to adoption.

Some latency-sensitive requests must be served from the cache, as they would otherwise miss their deadlines. Unfortunately, popular caching systems, such as Memcached, are deadline-oblivious and use a single LRU queue to manage cache eviction. In contrast, MicroFuge's cache eviction policy aims to minimize deadline misses. Instead of a single LRU queue, MicroFuge introduces a separate LRU queue for each deadline range. To make use of the multiple LRU queues, MicroFuge builds a performance model of the storage system using feedback from the clients. This model determines the likelihood that a particular cache eviction would lead to a deadline miss and is used to determine a cache eviction policy across queues.

Working in conjunction with the caching system is a distributed deadline-aware request scheduling layer that controls access to the cloud storage system. The scheduler keeps track of the pending requests of each storage server, directs client requests to storage replicas with lighter loads, and creates a latency model of the storage system to perform a variant of earliest deadline first scheduling while ensuring requests with unmeetable deadlines only minimally impact other requests.

Although an effective deadline-aware cache and scheduler can help meet performance requirements, it is nevertheless impossible to meet aggressive latency deadlines given an arbitrary request load. We address this problem in MicroFuge by adding an optional admission control component to our distributed scheduler. When activated, the scheduler uses its latency model of the storage system to provide early rejection of incoming requests that are unlikely to meet their latency requirements. This protects the storage system from being overloaded and ensures that a certain number of requests can still meet their performance requirements regardless of workload characteristics. The applications issuing these rejected requests can then make informed decisions on their next course of action.

An alternative to MicroFuge's middleware approach to performance isolation is to directly incorporate deadline-aware caching and scheduling into existing cloud storage systems. However, the current cloud storage ecosystem is greatly varied; there are dozens of systems that are widely in use today. Any effort to design a cache and scheduler for a specific system would only affect a small fraction of cloud applications and therefore will likely have minimal impact on future cloud application design.

The main contributions of this paper are as follows:

- The design and implementation of a deadline-aware, model-driven distributed caching system.
- A distributed scheduling system that performs a variant of earliest deadline first scheduling and request admission control.
- An evaluation that demonstrates the effectiveness of MicroFuge in an EC2 deployment using the YCSB [2] benchmark.

```
// Retrieve a value from the cache
CacheResult get(String key, double deadline);

// Insert a value into the cache
long put(double deadline, String key, String val,
    boolean overwrite, boolean isMissed);

// Remove a value from the cache
void erase(String key);
```

Listing 1: DLC interface functions.

The remainder of this paper begins with a description of MicroFuge's system architecture in Section II. Section III details our experimental setup and workloads, and Section IV presents our evaluation results. Section V surveys related work, and Section VI concludes.

II. System Architecture

The MicroFuge system is a multi-component middleware consisting of a distributed caching layer and a distributed scheduling layer. Both layers share a similar, scalable design in which coordination between servers is performed completely through communication between the servers and clients. This section describes the design of both layers, as well as the overall MicroFuge protocol.

A. Deadline Cache

MicroFuge's distributed caching layer, which we have named Deadline Cache (DLC), exposes a key-value store interface similar to that of Memcached. This interface is shown in Listing 1. The `get` operation fetches a key-value pair from the cache, and the `put` operation populates the cache, usually following a cache miss. Much like in Memcached, data stored in DLC is partitioned across the cache servers based on the hash of the key value, and DLC clients are provided the full list of cache servers at startup, which enables them to independently determine the cache server to contact for each request.

DLC's `put` operation includes a deadline field, which is used by the client to specify the maximum acceptable latency for servicing `get` requests for this key. It also includes a deadline missed field, which allows the client to specify if the `put` operation was performed in response to a cache miss which resulted in a deadline violation. This provides empirical feedback to the cache which is used to build a performance model of the underlying storage system. Both fields are used by the cache server to determine an appropriate eviction policy for a given deployment. DLC's `get` operation also includes a deadline field, which can be used to update the deadline for the specified data item.

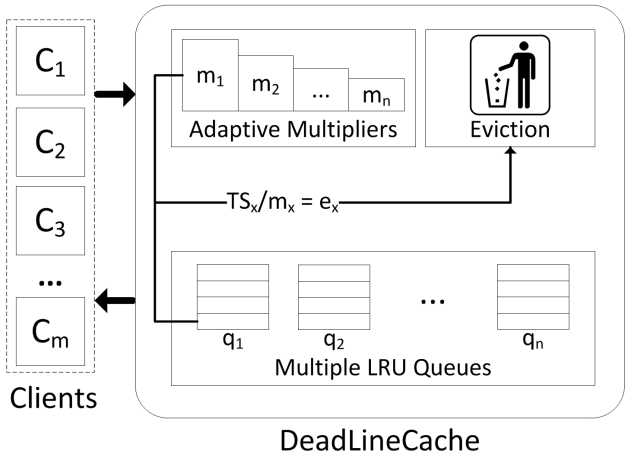

Figure 1: High-level layout of the DLC architecture.

The DLC server architecture is illustrated in Figure 1 and consists of three main components: a table for fast key lookups, multiple request queues for performing deadline-aware LRU eviction, and a performance modeling component that collects feedback from the clients to determine a cross-queue eviction policy in order to minimize deadline misses. We look at the latter two components in detail in the next subsections.

1) Multiple LRU Queues: Past work has shown that LRU-based cache eviction, where items are ordered by their last access time in a queue, offers near-optimal caching performance for general workloads [3]. Although LRU achieves high cache hit-rate and good performance, it is a deadline-oblivious eviction policy. 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 cache misses, DLC uses multiple LRU-ordered queues to manage cache eviction. Each queue is responsible for maintaining the relative ordering of key-value pairs with deadlines spanning a particular deadline range. The i-th queue is responsible for deadlines $[\frac{(i-1)\cdot D}{n}, \frac{i\cdot D}{n})$ where n is the number of queues in the cache server and D is the maximum deadline length. Items with deadlines larger than D are stored in the n-th queue. This multi-queue organization is illustrated in Figure 1.

The LRU queues ensure that items in the same queue are evicted in LRU order. However, determining which LRU queue to evict from to minimize the deadline miss rate requires knowing the likelihood that a particular eviction would eventually lead to a deadline violation. This determination is performed by DLC's performance modeling component.

2) Performance Modeling and Cache Eviction: DLC incorporates the cost of deadline misses into its eviction policy by applying a queue-specific multiplier $\frac{1}{m_i}$ to the difference between the current time and the last access time of an item. We call this product the Modified Recency Value (MRV) for each eviction candidate. The eviction candidates are the least recently used item from each queue, and the candidate with the largest MRV is selected for eviction.

An adaptive eviction policy must account for both the client request rate for each deadline range and the underlying storage system's performance. In a workload with requests that have a uniform deadline distribution, tuning a simple static multiplier assignment such that $m_{i-1} > m_i$ may be sufficient. However, such a scheme would perform poorly if only a very small percentage of requests have short deadlines. In this type of degenerate case, the short deadline data items will not be evicted even if they are only accessed infrequently, resulting in a low overall cache hit rate.

Similarly, an eviction policy that does not account for the underlying storage system's performance does not know whether the storage system can meet a given deadline under a particular client request load. Statically assigning a multiplier value that overestimates the performance of the storage system will lead to small multipliers for deadlines that the storage system cannot satisfy, resulting in additional deadline violations. Underestimating the storage system's performance will lead to apportioning more memory than necessary to store long deadline items. This will, in turn, lower the cache hit rate for short deadline requests (whose deadline satisfaction is much more cache hit rate-dependent), resulting in deadline violations. Therefore, it is critically important that a deadline-aware cache adapts its cache eviction policy based on deadline violation feedback from its clients, accounting for both client request rates and the storage system's performance.

MicroFuge's queue multipliers are adaptively computed using empirical measurements to reflect the likelihood that a cache eviction would eventually lead to a deadline miss. Its design follows that of a simple *proportional-integral* controller. In this design, the multipliers are initialized to 1 and the system maintains the following invariant:

$$\sum_{i=1}^{n} m_i = n \tag{1}$$

where n is the number of queues in the cache server. Upon receiving a `put` request with a deadline within the range of queue i and with the deadline missed flag set to true (indicating that a deadline violation occurred due to a cache miss), the multiplier m_i is incremented by ϵ. The value of ϵ affects the convergence rate and the amount of perturbation from new updates at steady state. In order to maintain the invariant in Equation 1, all of the queue multipliers are renormalized by multiplying by $\frac{n}{n+\epsilon}$. The relative increase in m_i reflects the increased number of observed deadline misses in the deadline range of queue i.

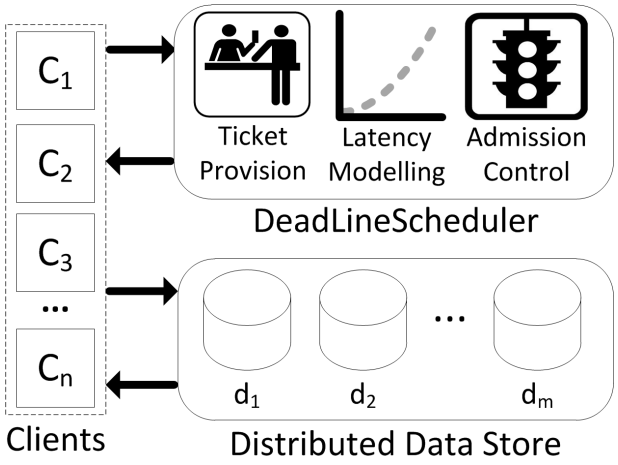

Figure 2: High-level layout of the DLS architecture.

B. Deadline Scheduler

The second major component of MicroFuge is its distributed scheduling layer, the Deadline Scheduler (DLS), which is typically deployed on the same servers as DLC, and collectively tracks the outstanding requests of each storage server. The DLS interface is shown in Listing 2 and provides a ticket-based approach to perform reservations. By serving as an intermediary between the clients and the storage servers, it can perform load-balancing of requests across replicas, control the ordering of client requests to reduce the number of deadline violations, and optionally perform admission control to provide early rejection of requests that cannot make their deadlines. Three mechanisms work in unison to provide DLS with its capabilities. These mechanisms are:

- A ticket-based load-balancing system that directs each client to the server that is the most likely to be able to meet the request's deadline.
- A variant of earliest deadline first scheduling that uses performance statistics from the underlying storage system to limit the impact of deadline violations on other requests.
- A tunable admission control system that uses performance statistics to minimize the number of rejections while meeting a deadline miss rate target.

We describe each of these mechanisms in turn in the following sections. A high-level depiction of clients interacting with the scheduling layer and the data store can be seen in Figure 2.

1) Ticket-Based Load-Balancing: Most cloud storage systems [4], [5], [6] perform data replication to provide fault tolerance, increase availability and improve read performance. The amount of flexibility available in replica selection for reads depends on the storage system's consistency model. Systems that offer strong consistency may require that the primary replica service all of the read operations. In contrast, eventually consistent systems, which make up the majority of current cloud storage systems, can service reads from any replica, with most of these systems using some type of randomized selection technique. Unfortunately, randomized selection can lead to unpredictable hotspots which may cause deadline violations.

MicroFuge leverages the replica selection flexibility in current cloud storage systems to both improve load balancing and reduce deadline violations. The basic approach follows the load balancing algorithm proposed by Mitzenmacher [7] in which a client that wishes to issue a read request will randomly choose two of the replicas as potential read candidates. It will then send ticket requests to the DLS servers responsible for these storage servers, where each ticket represents a read reservation. The tickets will include the deadline for the read request, and the DLS servers will determine, based on the number of outstanding requests to the server and the server's performance model, whether or not it believes the storage server can service the request within its deadline. Unlike in Mitzenmacher [7] where the less loaded of the two servers is always selected, the client will first select a server that can satisfy its request deadline while not causing someone else to miss their deadlines and only use system load to render a decision if both of the servers are either able or unable to satisfy the request deadline.

After selecting one of the replicas, the client cancels its ticket on the other replica, and sends a `waitOnTicket` request to the scheduler to wait for its turn to access the storage system. Upon receiving a response from the `waitOnTicket` request, the client can issue its read request to the storage server. Once the read completes, the client sends a final `releaseTicket` request to DLS to notify the scheduler that the request has completed. DLS uses the `releaseTicket` request to both determine when it can allow the next client to issue its request to the storage server and build a performance model for the response times of the read requests.

2) Scheduling Algorithm: DLS uses a variant of earliest deadline first (EDF) scheduling to determine the ordering of pending requests (tickets)[1]. EDF is known to be optimal for single-resource scheduling with preemption if a schedule exists where all of the request deadlines can be met [8]. However, in overloaded situations where not all deadlines can be met, EDF scheduling can lead to additional deadline violations by attempting to schedule requests that either cannot meet their deadlines or have already missed their deadlines.

The DLS variant of EDF examines the response time performance model of the storage server in order to determine whether a pending request should be scheduled using

[1]We would like to point out that DLS provides write monotonicity. In other words, writes are never reordered by the scheduler.

```
// Request a ticket from the scheduler
int getTicket(enum opType, double deadline,
    boolean bestEffort);

// Wait on a ticket until it is scheduled to
    proceed
boolean waitOnTicket(int ticketId, enum opType,
    int dbServerId);

// Cancel a request and remove it from the queue
void cancelTicket(int ticketId, enum opType,
    int dbServerId);

// Remove a completed request and update latency
void releaseTicket(int ticketId, double
    dbLatencyMs,
    enum opType);

// Combine the get and wait actions (best effort)
boolean getTicketAndWait(enum opType,
    double deadline);
```

Listing 2: DLS interface functions.

its specified deadline, or rescheduled using a much larger artificial deadline in the case where a deadline violation is inevitable. By rescheduling these requests with a larger deadline, despite the fact that these requests will miss their own deadlines, DLS enables other requests to be scheduled earlier and increases the likelihood of them meeting their deadlines. To prevent starvation, DLS only allows a request to be rescheduled once.

DLS's response time performance model uses request latencies from a past window of requests to generate a latency distribution. The request latencies are provided by the clients as part of the `releaseTicket` operation. DLS determines that a request is unable to meet its deadline if the time remaining to meet its deadline is less than the α-percentile request latency in the latency distribution of the response time performance model, where α is a system parameter. The remaining time is calculated using the difference between the request deadline and either the current time if the storage server is idle or the estimated completion time of the previous request using the performance model.

3) Request Admission Control: MicroFuge is primarily designed to prevent deadline violations. Deadline-awareness in the cache eviction policy helps reduce the chance of misses by increasing the cost of cache evictions that are likely to cause deadline violations. The ticket-based scheduling algorithm helps distribute load and further reduce deadline misses by both scheduling earlier deadline requests ahead of later deadline requests and rescheduling requests that will inevitably cause deadline violations to minimize their impact on other requests' response times. Despite all this, as the load on the system increases, deadline misses are unavoidable, and previous work has shown that outstanding requests on a data store can significantly impact an application's response time [9]. Therefore, to satisfy the performance requirements of at least a subset of the requests, a portion of new requests must be rejected by an admission control system.

DLS provides an optional admission control mechanism that uses the performance model described in Section II-B2 to determine, given the current list of pending requests, if the new request will likely miss its deadline or cause one of the pending requests to miss its deadline due to the EDF scheduling policy. In either case, if the admission control is enabled, DLS will reject the `getTicket` operation instead of returning a ticket. The admission control mechanism uses the β-percentile latency in the latency distribution to estimate the completion time of pending requests, where β is a tunable parameter. A higher β value will lead to fewer deadline violations and a higher rejection rate, while a lower β value has the opposite effect.

By allowing applications to define their own deadlines on storage requests, MicroFuge allows the hosting of applications to cater to clients with a variety of latency needs. Performance is bounded by client-defined performance metrics, and not the generic (and possibly unhelpful) decisions of the scheduler inside the data store itself. 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. For example, a web application client can, upon receiving a rejection from both tickets, serve a static advertisement, which does not require a read request to the storage system, instead of displaying a list of recommended or related items.

C. MicroFuge Protocol

The MicroFuge request protocol combines both sequential and concurrent requests to DLC, DLS and the underlying storage system. A MicroFuge read request begins with the client issuing a key lookup request to the DLC. The request completes if the key is available in the cache. Otherwise, the client randomly selects two storage servers with a copy of the request data item, and issues a `getTicket` request to each of the DLS servers that are managing the pending requests for these storage servers.

The client waits until it has received a response from each `getTicket` request. Responses returned may be marked as successful, which means the scheduler believes that the storage server will service the request within the request's deadline and, if the admission control is enabled, will not cause other pending requests to miss their deadlines, or unsuccessful, which means the request will likely miss its deadline. Responses also include an indication of how long the scheduler believes it should take to service the request. If only one of the two responses are marked successful, then the storage server with the successful response is selected. If both responses are marked as unsuccessful and admission control is enabled, then the request is rejected. Otherwise,

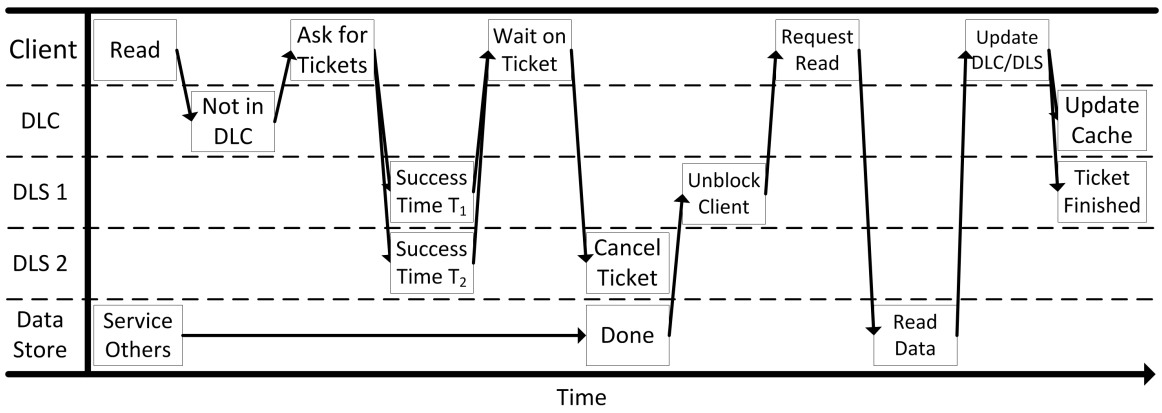

Figure 3: Sample timeline for a read request from a client. For this request, the requested item is not contained in the cache and both schedulers accept the ensuing ticket requests.

```
// Retrieve a value from MicroFuge
public String read(String key, double deadline,
    boolean bestEffort);

// ... Other code here

String myVal = read("myKey", 12345, true);
```

Listing 3: MicroFuge `read` operation interface.

the storage server with the shorter expected service time is selected.

After selecting a storage server, the client concurrently cancels the ticket for the unselected server, and issues a `waitOnTicket` operation to the scheduler for the selected server. Once the `waitOnTicket` operation completes, the client can then issue a read request to the storage server to retrieve the requested data item. Finally, upon receiving the result from the storage server, it concurrently issues a `releaseTicket` operation to the scheduler and a `put` operation to the cache to populate this data item.

Managing the interactions between DLC, DLS, clients and storage systems can be complicated. These systems can be neatly encapsulated into a very simple and easy-to-use client protocol, exposing only operations such as `read` and `write` to the user. As an example, the MicroFuge interface that exposes a read operation can be seen in Listing 3. This interface performs all the necessary operations to retrieve a value from DLC and DLS.

We illustrate the request protocol in Figure 3, which outlines the steps taken to perform a read request. This example assumes that the data is not in the DLC and the data store is busy processing another request when the sample request is issued. After contacting the appropriate scheduling nodes, the client is informed that both scheduling nodes are successfully able to complete its request. The client selects the first scheduling node (which reports an earlier completion time estimate than the second node), cancels its request to the second node and then waits on notification from the storage layer before performing its read. Immediately after the completion of its read, the sample request releases its ticket, concurrently updates the DLC with the data value and the scheduler with the latency information.

III. Experimental Setup

We deployed our system on a twenty-node test cluster on Amazon Web Services. Each cluster node is an m1.medium EC2 instance with two elastic compute units, 1 virtual CPU, 3.7 GB memory, 410 GB of (non-EBS) storage and moderate network performance. All nodes run 64-bit Ubuntu Server 12.04.3 and we manually set the memory size for each node to 2 GB to cut down on the time required to warm up the cache for the experiments. Four instances were configured as clients to run YCSB benchmarks. The rest of the (sixteen) instances were configured as servers, each running MicroFuge or Memcached on top of our cloud storage system. All machines are in the same subnet within AWS's network which would be representative of a datacenter setup.

Our data set consists of 80 million records 86.4 GB in size. The total cache capacity of our system is 19.2 GB, about 1/5th the size of the data set. The data was stored in a simple custom data storage system based on leveldb [10].

We used YCSB [2] to generate our workloads. Each request generated by YCSB was for a 13-byte key associated with a 1 KB value. We modified YCSB to generate a deadline for each request. Deadlines are generated using the hash of the key so requests for the same key will always have the same deadline. This represents the scenario where each key is associated with a particular application, and, for a given key, the application uses the same request deadline. The generated deadlines fall into one of 3 ranges of deadlines: [10-30) milliseconds, [30-100) milliseconds and [100-1000] milliseconds with a distribution ratio of 2:3:5 respectively. These ranges essentially represent classes of clients where each class has a fixed range of response time requirements.

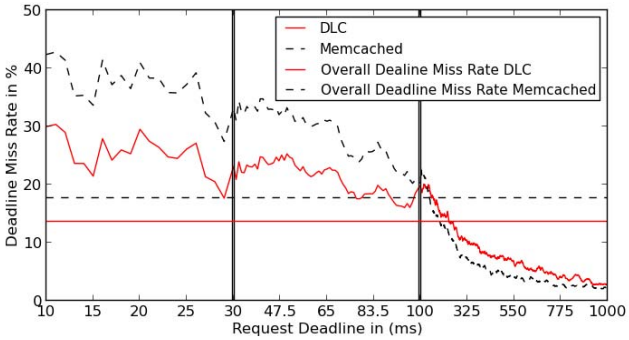

Figure 4: Deadline miss rate for 192 concurrent clients with DLC and Memcached.

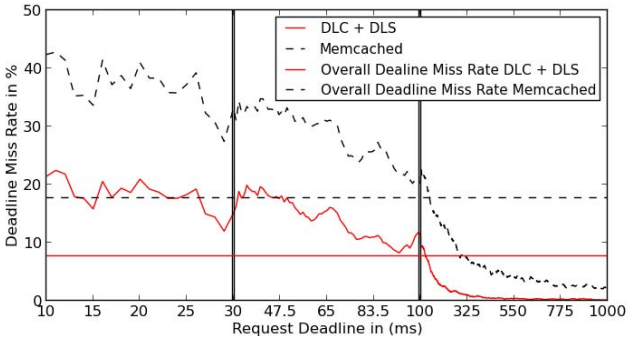

Figure 5: Deadline miss rate for 192 concurrent clients with DLC + DLS and Memcached.

Each data point on the performance graphs in the next section is the average value of 5 independent runs after the system has warmed up with the system parameters α and β set to default values of 15 and 88 respectively. Both values are determined by empirical experimentation. A small α value enables the underlying storage system to aggressively schedule client requests so it can utilize the system resources at full capacity while a large β value will minimize the deadline miss rate in order to satisfy requests with very strict service level objectives. The 95th percentile confidence intervals on the relevant graphs are shown as error bars around the data points.

IV. Performance Evaluation

The first goal of our experiments is to compare the deadline miss rates of our adaptive deadline-aware caching and scheduling layer running on top of our simple data storage system against Memcached running with the same storage system. The second goal is to show that our cache is deadline-aware, retaining data items with shorter deadlines in the cache by evicting data items with longer deadlines. The third goal is to illustrate the tunable admission control mechanism, which can further bound the deadline miss rate of the underlying storage system to a desired upper limit. Lastly, we want to show a comparison of the overall deadline miss rates and overall cache hit rates of the different components described in this paper while running the YCSB workload. Where appropriate, we include measurements for our storage system as a baseline comparison. Note that the baseline storage system refers to simply using the storage layer without any caching, scheduling or admission control components.

Figures 4, 5 and 6 show the deadline miss rates for requests with varying deadlines. Figure 4 shows that MicroFuge's caching layer does better than Memcached in reducing deadline misses for requests with shorter deadlines. Our caching layer favours requests with shorter deadlines since longer deadline requests are unlikely to be missed. Our

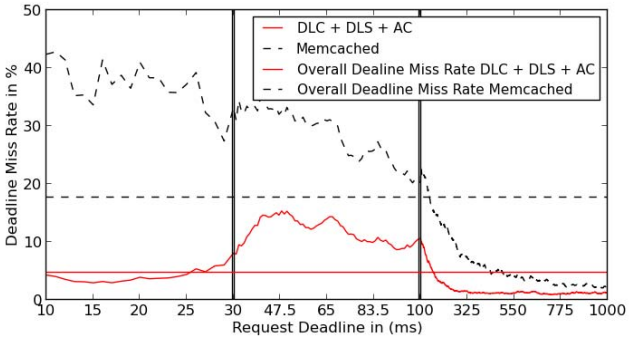

Figure 6: Deadline miss rate for 192 concurrent clients with DLC + DLS + AC and Memcached.

scheduler, which adds adaptivity to MicroFuge, balances the load of the storage system so that the deadline miss rate for requests with long deadlines is also reduced as seen in Fig. 5. If we turn on our admission control inside MicroFuge's scheduling layer, we can further reduce overall deadline miss rates to less than 5%, which is almost a 74% improvement over Memcached's overall deadline miss rate for 192 clients. (This deadline miss rate does not include the 7.14% rejection rate.) As seen in Figure 6, the scheduler rejects any requests with short deadlines that are unlikely to be met. This is part of our design, as we would like to inform the client without any further delay that we are unlikely to meet its deadline, thereby allowing the client to decide on the next most desirable course of action.

Figures 7, 8 and 9 show the cache hit rates for both MicroFuge and Memcached. MicroFuge's overall cache hit rate is marginally lower than that of Memcached. However, our results also demonstrate that our system is deadline-aware, as we tend to keep items with lower deadlines in the cache. Memcached, which has no deadline-awareness, has an almost uniform cache hit rate across different request deadlines. By using both DLC and DLS, additional requests

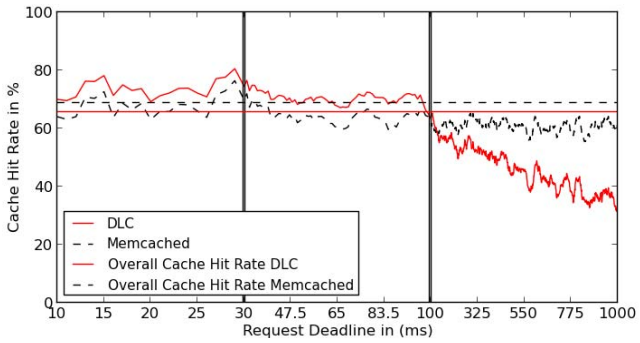

Figure 7: Cache hit rate for 192 concurrent clients with DLC and Memcached.

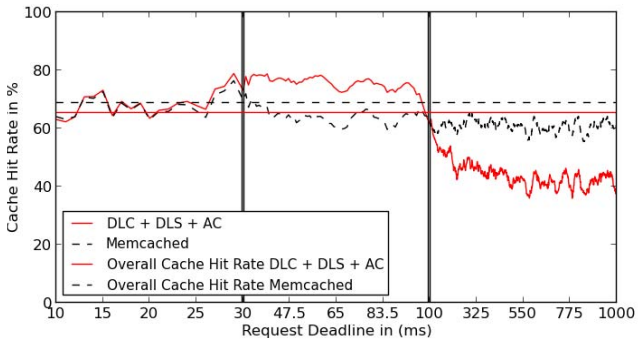

Figure 9: Cache hit for 192 concurrent clients with DLC + DLS + AC and Memcached.

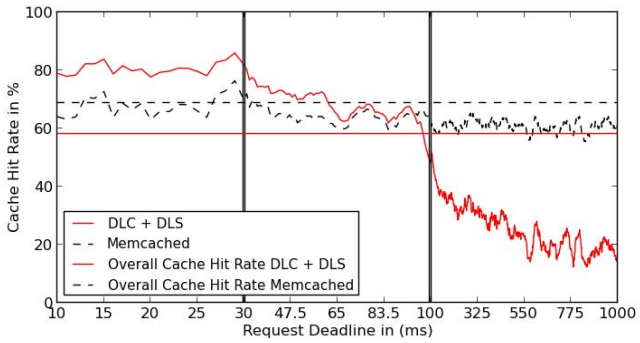

Figure 8: Cache hit rate for 192 concurrent clients with DLC + DLS and Memcached.

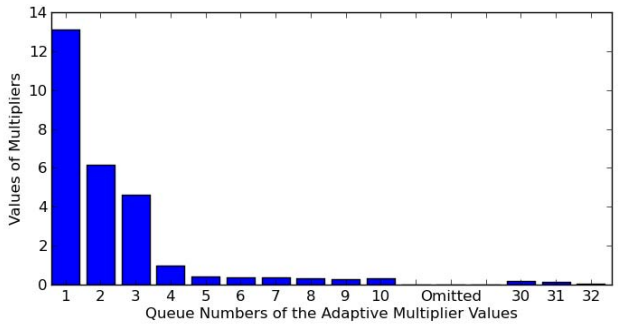

Figure 10: A snapshot of the converged adaptive multipliers for 192 concurrent clients with DLC only.

with long deadlines can be satisfied from the disk compared to just using DLC. Therefore, there is a reduced need to cache items with long deadlines, which leads to more resources being available to cache items with short deadlines and a higher cache hit rate for requests with deadlines less than 30 ms as shown in Figure 8. In Figure 9, with admission control enabled, there is a lower cache hit rate for deadlines less than 30 ms. This is because short deadline items are often rejected and therefore are less likely to be inserted into the cache. Requests with deadlines between 30 ms and 100 ms have a higher cache hit rate as they are less likely to be rejected and there are more resources available due to the low occupancy of short deadline items. Lastly, Figure 10 shows a snapshot of DLC's underlying performance model where there is a multiplier associated with each queue, as described in Section II-A2. Our performance model favours requests with shorter deadlines as they are given much larger multipliers, increasing the likelihood that these requests are kept in the cache. Although we omit some adaptive multipliers due to space constraints, these omitted values follow the same trends shown in Figure 10.

Figure 11 shows that our tunable admission control mechanism can reduce overall system deadline miss rates to as low as 3%-4%. By varying the system parameter β, the sum of the rejection rate and deadline miss rate is approximately the same as the deadline miss rate for DLC and DLS without admission control. This gives the cloud service provider a useful knob that can be varied to protect servers against overloading.

The graphs in Figures 12 and 13 demonstrate the overall deadline miss and cache hit rates for various system setups. They show that each of MicroFuge's components contribute to reducing deadline misses. The overall cache hit rate is only marginally lower than Memcached's cache hit rate due to the non-uniform cache eviction policy of MicroFuge which shows that MicroFuge's cache is as effective as Memcached's. Figures 12 and 13 also show that a 10% write and 90% read workload has minimal impact on the overall deadline miss rate and overall cache hit rate. The deadline miss rate of our simple cloud storage layer included as a baseline is over 20% at just 96 clients. This depicts the significant performance degradation that a storage system can suffer from if a concerted effort is not made to provide deadline-aware mechanisms such as caching, scheduling and

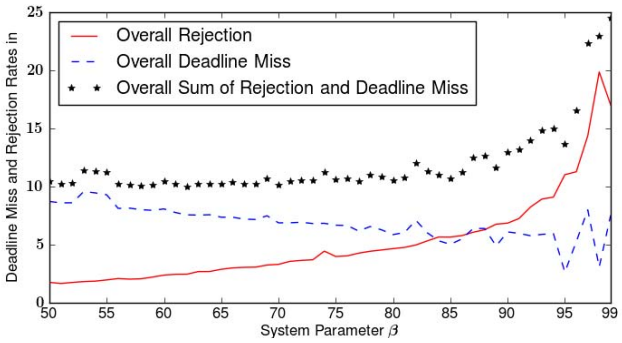

Figure 11: Deadline miss vs. rejection rates with respect to various values of system parameter β for 192 clients.

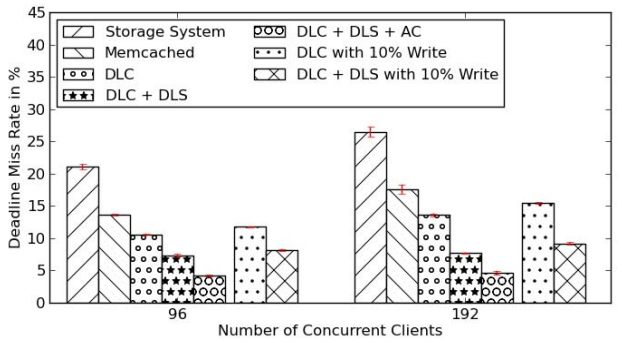

Figure 12: Overall deadline miss rate with various system setups for 96 and 192 concurrent clients.

admission control.

Based on our experimental results, MicroFuge's caching layer outperforms Memcached by 22.6% for 192 clients. If we combine our scheduling layer with our caching layer, we can further decrease the deadline miss rate by 43.3%. If we turn on our admission control, we can keep the overall deadline miss rate below 5%.

V. Related Work

Previous work has examined cache sharing and scheduling for multi-tenant systems. The Argon storage system [11] shares some similarities with MicroFuge. It 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, however, but 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

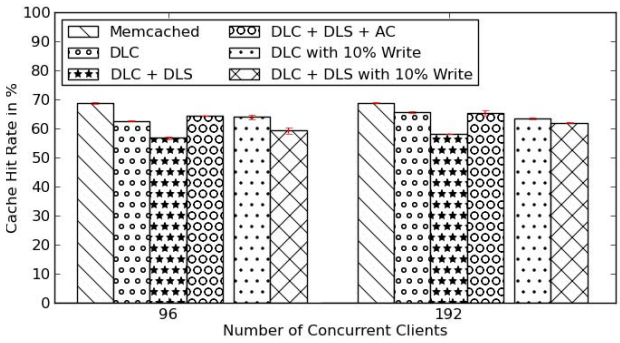

Figure 13: Overall cache hit rates with various system setups for 92 and 192 concurrent clients

sharing, whereas MicroFuge makes explicit considerations for deadline-based requests when scheduling through the use of latency metadata collection. Furthermore MicroFuge directly addresses performance isolation through its multiple deadline-oriented LRU queues, and its eviction policy that favours data with shorter deadlines.

Work similar to our own is the Frosting system [12], which 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 attempts to bound outstanding requests while minimizing queuing at the data store layer (in an effort to reduce response times). In a manner more similar to Argon than MicroFuge, however, Frosting focuses primarily on system throughput and fairness, not on strict performance isolation.

The Sparrow system acts as a decentralized, stateless scheduler and is geared towards workloads with high degrees of parallelization in addition to workloads that require low latencies [13]. Sparrow demonstrates that high volumes of low priority tasks can have detrimental effects on the scheduling response times of high priority tasks, and that their system can greatly improve median scheduling response times. Sparrow differs from MicroFuge, however, in that it focuses on minimizing overall scheduling response time in a system, and is not concerned with accommodating and meeting user-imposed scheduling deadlines per request.

Memcached [14] is designed 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 provides no admission control.

MemC3 [15] proposes the use of optimistic hashing with CLOCK-based cache management to improve 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 similarly modifies Memcached to provide inter-VM shared memory [16], with a good degree of success and the potential for complementary usage of its presented techniques with other caching systems. Unlike MicroFuge, however, Nahanni is limited to VMs running on a physical host and provides almost no performance isolation guarantees.

Pisces [17] proposes a group of mechanisms for partitioning resources between users. The paper suggests that by considering partition placement, weight allocation, replica selection and fair queuing for resources, the system can split aggregate throughput in the system between clients. Although Pisces does provide throughput isolation for performance, its scope does not extend to the deadline and latency-aware mechanisms that MicroFuge uses to provide performance isolation.

The FAST system [18] introduces a block-level replicated storage service that helps provide performance predictability by overlapping similar operations (sequential reads versus random writes, for example) on the same machines, which minimizes interference. Unlike MicroFuge, FAST's primary focus is on system fairness, and it additionally does not consider explicitly-defined request deadlines.

In the BASIL system, 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 focus on cloud data stores and cloud-like access patterns. Furthermore, BASIL 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 [19] proposes 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 also 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 [4], [20], [21]. These key-value stores, designed with the cloud in mind, can often suffer from poor performance isolation.

Performance modeling in datacenters is increasingly common as more applications move into the cloud. Data Center TCP [22] presented a model of underlying network traffic patterns inside datacenters, and proposed a new datacenter-oriented version of TCP communication to help deal with the problems typically associated with these patterns. iCBS [23] presents a method for quickly determining effective orderings for generic requests given arbitrary SLA cost functions, but does not deal with deadline-specific concerns within the context of a distributed key value store.

There are many approaches to performing both external and internal scheduling for admission control. Schroeder et al. [24] considers optimizing concurrency levels in database systems through admission control. Abbott and Garcia-Molina [25] propose models for performing admission control aimed at real-time database systems using deadlines. They use simulations to understand the performance tradeoffs of utilizing transactional commit behaviors 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.

VI. Conclusion

In this paper we introduced MicroFuge, a new middleware layer 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 obtain inside multi-tenant systems. MicroFuge is built with a similar API as Memcached, and is easy to layer on top of cloud storage systems.

MicroFuge's distributed caching layer, DLC, uses an adaptive and deadline-aware cache eviction policy to isolate performance and ensure that additional caching memory is allocated towards requests which are more likely to miss their deadlines. This is performed with the use of multiple LRU queues based on deadline ranges in combination with adaptive policies.

MicroFuge's distributed scheduling layer, DLS, uses a ticket-based scheduling system that helps to not only balance load, but also to service requests according to their deadline requirements. Additionally, 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 Memcached, the current industry standard. With admission control enabled, MicroFuge can limit overall deadline miss percentages to less than 5%.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments. This work is supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] P. Patel, A. Ranabahu, and A. Sheth, "Service level agreement in cloud computing," in *Cloud Workshops at Object-Oriented Programming, Systems, Languages & Applications*, ORLANDO, FLORIDA, USA, 2009.

[2] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *ACM Symposium on Cloud Computing*, Indianapolis, Indiana, USA, 2010.

[3] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "An optimality proof of the LRU-*K* page replacement algorithm," *Journal of the ACM*, vol. 46, 1999.

[4] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *Operating Systems Review*, vol. 44, 2010.

[5] "Apache HBase," http://hbase.apache.org.

[6] "MongoDB," https://www.mongodb.org/.

[7] M. Mitzenmacher, "How useful is old information?" *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, 2000.

[8] T. Baker, "Deadline scheduling," http://www.cs.fsu.edu/~baker/realtime/restricted/notes/edfscheduling.html.

[9] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar, "Basil: Automated IO load balancing across storage devices," in *USENIX Conference on File and Storage Technologies*, San Jose, California, USA, 2010.

[10] Google Inc., "leveldb: A fast and lightweight key/-value database library by google." http://code.google.com/p/leveldb/.

[11] M. Wachs, L. Xu, A. Kanevsky, and G. R. Ganger, "Exertion-based billing for cloud storage access," in *USENIX Workshop on Hot Topics in Cloud Computing*, Portland, Oregon, USA, 2011.

[12] A. Wang, S. Venkataraman, S. Alspaugh, I. Stoica, and R. Katz, "Sweet storage SLOs with frosting," in *USENIX Workshop on Hot Topics in Cloud Computing*, Boston, Massachusetts, USA, 2012.

[13] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *ACM Symposium on Operating Systems Principles*, Farmington, Pennsylvania, USA, 2013.

[14] "memcached - a distributed memory object caching system," http://memcached.org.

[15] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing," in *USENIX Symposium on Networked Systems Design and Implementation*, Lombard, Illinois, USA, 2013.

[16] A. W. Gordon and P. Lu, "Low-latency caching for cloud-based web applications," in *Workshop on Networking Meets Databases*, Athens, Greece, 2011.

[17] D. Shue, M. J. Freedman, and A. Shaikh, "Fairness and isolation in multi-tenant storage as optimization decomposition," *ACM SIGOPS Operating Systems Review*, vol. 47, 2013.

[18] X. Lin, Y. Mao, F. Li, and R. Ricci, "Towards fair sharing of block storage in a multi-tenant cloud," in *USENIX Workshop on Hot Topics in Cloud Computing*, Boston, Massachusetts, USA, 2012.

[19] V. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri, "SQLVM: Performance isolation in multi-tenant relational databases-as-a-service," in *Biennial Conference on Innovative Data Systems Research*, Asilomar, California, USA, 2013.

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

[21] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *IEEE Symposium on Mass Storage Systems and Technologies*, Incline Village, Nevada, USA, 2010.

[22] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," *ACM SIGCOMM computer communication review*, vol. 41, 2011.

[23] Y. Chi, H. J. Moon, and H. Hacigümüş, "iCBS: incremental cost-based scheduling under piecewise linear SLAs," *Proceedings of the VLDB Endowment*, vol. 4, 2011.

[24] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter, "Priority mechanisms for OLTP and transactional web applications," in *IEEE International Conference on Data Engineering*, Boston, Massachusetts, USA, 2004.

[25] R. K. Abbott and H. Garcia-Molina, "Scheduling I/O requests with deadlines: A performance evaluation," in *IEEE Real-Time Systems Symposium*, Lake Buena Vista, Florida, USA, 1990.