Hoover: Overload Control for Applications with Unpredictable Lock Contention

Paper #326

Abstract

Modern datacenter applications are concurrent, so they require synchronization to control access to shared data. Requests can contend for different combinations of locks, depending on application and request state. In this paper, we show that locks, especially blocking synchronization, can squander throughput and harm tail latency, even when the CPU is underutilized. Moreover, the large number of contention points and the unpredictability in which locks a request will require make it difficult to prevent contention through overload control using traditional signals such as latency and CPU utilization.

We present Hoover, a system that resolves these problems with two key ideas. First, it introduces new latency-aware blocking synchronization abstractions that allows applications to abort requests if locking delays exceed latency objectives. Second, it contributes a new overload control strategy that prevents compute congestion in the presence of lock contention. The key idea is to use marginal improvements in observed throughput, rather than CPU load or latency measurements, within a credit-based admission control algorithm that regulates the rate of incoming requests to a server. We apply Hoover to two real-world applications, Memcached and Lucene, and show that it achieves up to 6.2× goodput with 140× lower and bounded 99th percentile latency complying with SLO.

1 Introduction

One of the key objectives of datacenter operators is to maximize the utilization of limited resources. While operating a server close to its capacity maximizes its throughput, it also makes it susceptible to overload due to surges in demand. Such surges can occur due to variability in request arrival patterns and sizes, and service failures. The resulting server overload can cause *receive livelock*, where the server builds up a long queue of requests that get starved because the server is busy processing new packet arrivals instead of completing pending requests [16].

The conventional solution is to use *overload control* to shed or prevent excess load in order to ensure the server can achieve both high utilization and low latency. Existing overload control schemes rely on metrics like request latency and CPU utilization to manage and throttle request rates [7,21,23]. These approaches work well in overload scenarios where the CPU is the contended resource. However, we found they perform poorly under lock contention, either underutilizing CPU resources or permitting long queues to form as requests wait to acquire locks(§2).

Driven by increasing core counts, most applications have significant amounts of shared data, and must use fine-grained locks (or other synchronization primitives) to maintain good multicore scalability. An incoming request that only accesses one piece of data may need to acquire just a single lock. However, it is hard to predict which lock a request will need to acquire until that request is actually processed.

For example, consider a key-value store, where items are grouped together based on their hashes. Access to a bucket (i.e., a group of items with the same hash) is protected by an item lock. This means that in a key-value store, the number of locks corresponds to the number of buckets. However, a `GET` request would require access to only a single bucket based on its request payload.

When the lock protecting this bucket becomes highly contended, it will negatively impact the latency of some but not all of the requests the application handles. However, managing overload for this locking object is difficult. For example, it can become overloaded even if the overall request rate is insufficient to saturate the CPU. Moreover, when this occurs, it may still be necessary to continue admitting additional requests to increase the CPU utilization.

In this paper, we attempt to answer the following question: *how should an overload controller decide to admit a request when it can't estimate the delay the request will face?* Tackling this challenge is exacerbated when considering that some applications can have thousands of locks. Further, shedding load after processing a request requires cleaning up the state and resources touched by that request.

We present Hoover, a system that provides overload control for applications that can experience lock contention. It solves

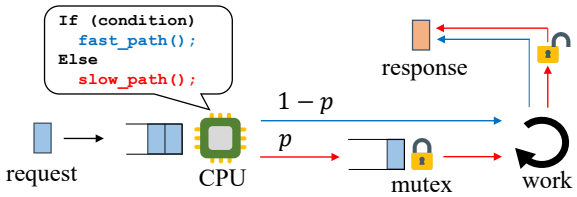


Figure 1: A simple example application with a mutex. With a probability p, the request follows slow path (red arrows).

this problem through two key ideas(§3). First, it introduces new latency-aware synchronization primitives that allow applications to differentiate requests and abort them when lock contention is too severe. Second, it provides a new overload control algorithm that relies on observing marginal changes to throughput in response to load changes, and a credit-based admission control policy to regulate the incoming request rate to the server. As a result, it can find the optimal offered load to maximize a server's goodput, even if some requests must be aborted during processing.

We implemented Hoover by extending and modifying Breakwater(§4). We additionally implemented latency measurement of the queues for mutex and conditional variable, per-request latency tracking feature, and latency-aware synchronization APIs, and modified credit management logic in Breakwater so that it adjusts the credit pool size based on the measured throughput.

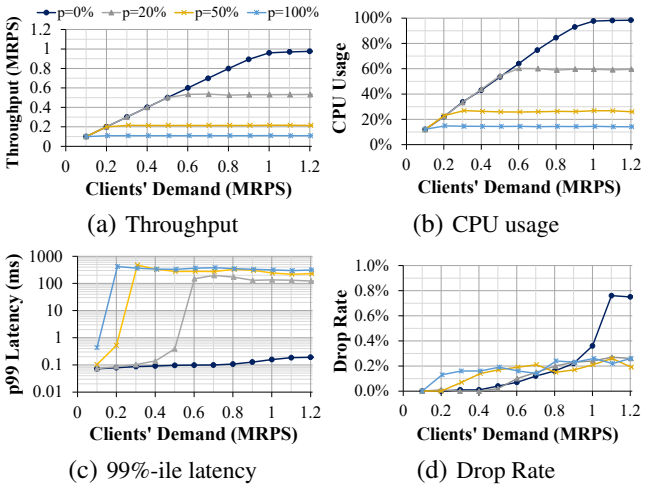
A limitation of Hoover is that it requires application-level code changes to adopt our synchronization API. However, our evaluation demonstrates that Hoover outperforms Breakwater, a state-of-the-art overload control, for a wide range of workload and applications(§5). For example, when Memcached is handling a SET-heavy workload, Hoover achieves up to 6.2× goodput with 140× lower 99th percentile latency than Breakwater.

2 Motivation

2.1 Detecting Overload with Synchronization

In modern datacenter applications, requests require synchronization (e.g. locks, mutexes, conditional variables, etc.) to serialize access to shared data. However, synchronization primitives often experience contention with other concurrent requests and can become a performance bottleneck. Unfortunately, lock contention cannot be easily detected by traditional methods. For example, CPU utilization, which is a widely used congestion signal, no longer works for detecting a mutex since threads that are queued waiting to acquire a mutex don't consume CPU cycles. Moreover, the locks used by a request may differ by each request depending on the request's data or state. This unpredictability makes overload control even harder.

To understand the problem better, let's imagine a simple application where it is difficult to predict which mutex each request will acquire (shown in Figure 1). When a request arrives to the server, the network packet is handled by a request thread by a dispatcher[1], and it is enqueued to the thread queue waiting for the CPU for execution. Once a request gets scheduled to a CPU, the request may go through the slow data path with probability p ($0 \leq p \leq 1$) where it must acquire a mutex while doing a work; otherwise, it goes through the fast data path where the request can do work without holding the mutex. In this simple application, all requests busy-loops for a specified amount of time to mimic the work done by real requests.



(a) Throughput (b) CPU usage

(c) 99%-ile latency (d) Drop Rate

Figure 2: Performance of Breakwater with simple application with a mutex contention

When the mutex becomes congested in this example application, how could we determine whether the application is overloaded or not? There are three commonly used overload signals: CPU usage [3], request execution delay [7, 23], and end-to-end latency [4, 10, 13, 14, 17, 21]. However, none of them work effectively in this example. Under contention in a mutex, a CPU usage or request execution delay signal leads to high tail latency because the signal does not capture the queueing buildup at the mutex. As the signal cannot be used to back-pressure the incoming load, requests in the slow path suffer from high mutex queueing delay. On the other hand, an end-to-end latency signal can capture the mutex queueing delay, but unpredictable requirement for mutexes makes latency signals too conservative, which leads to CPU under-utilization. While the high tail latency is only caused by the slow path, it tries to limit the incoming load so that mutex queueing delay does not grow too much, overly limiting throughput in the fast data path.

To demonstrate these problems with existing overload control signals, we ran the application shown in Figure 1 using Breakwater, a state-of-the-art overload controller [7]. Breakwater controls overload based on the request execution delay with credit-based admission control and AQM. We ran experiments varying p from 0% to 100% while 1,000 clients

[1]Throughout the paper, we assume dispatcher threading model where the server spawns a thread for each request because the dispatcher model make latency tracking for each request easier with low overhead [7]

generate requests with an open-loop Poission arrival process. Each request spins for a time sampled from exponential distribution with a 10µs average. When a request starts to execute, it takes either the fast path or slow path based on its hash value.

Figure 2 reports the throughput, server's CPU usage, 99-percentile latency, and drop rate. Because only one thread can acquire the mutex at a time, the slow path's capacity is equivalent to the capacity of only one core. An application without the mutex ($p = 0\%$) can achieve 1M reqs/s with 10 cores, so the slow path's capacity is 100k reqs/s($p = 100\%$).

When p is less than 10%, the CPU becomes congested before the mutex, so Breakwater effectively controls the overload, and provides high throughput and low and bounded tail latency. However, the mutex becomes contended with p larger than 10%, as the load to the slow path exceeds 100k reqs/s, causing the mutex's queue to keeps growing ending until a large number of requests are waiting to acquire the mutex.

Because threads do not consume CPU while waiting for the mutex, CPU usage remains less than 100% and request execution delay remains small. As a result, Breakwater keeps issuing a new credit up to the maximum[2], and AQM in Breakwater rarely drops the requests.

Once the credit pool size reaches its maximum, the server only issues a new credit when it completes a request and send back a response. With a newly issued credit, a new incoming request will go through fast data path $(1-p)/p$ times on average before getting stuck in the mutex queue in the slow path again, achieving the fast path throughput of $((1-p)/p)$ times of the slow path throughput (100k reqs/s), under-utilizing the fast path. Since the higher the p, the longer the mutex queue, 99th percentile latency grows with higher p. Real-world data center applications are more complicated than the simple example application we have shown, as they typically have more synchronization and unpredictability, making overload control an even more challenging task.

2.2 System Objectives

An ideal overload controller should provide the following properties, even for applications with lock contention:

1. High goodput. An RPC server should provide the high throughput of the requests that meet its SLO (we define it as goodput) which requires both high throughput and low tail latency.

2. Low drop rate. As far as the RPC server achieves the same goodput, it should minimize the drop rate to reduce the expected number of retires to achieve target success rate within a service-level SLO.

3. Minimal number of parameters. Large number of parameters are huge overhead for the service operators to tune. To minimize such a tuning overhead to deploy overload control system, the system should have a minimal number of parameters.

[2]Most RPC server limits the number of pending requests at the server to bound the maximum memory usage by RPC requests. In Breakwater, with the maximum number of 64 credits per connection by default, it can accommodate up to 64,000 requests with 1,000 clients.

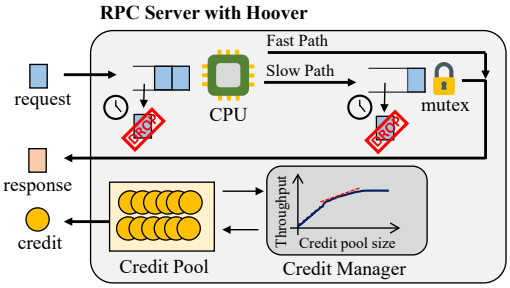


Figure 3: Hoover Overview

2.3 Challenges

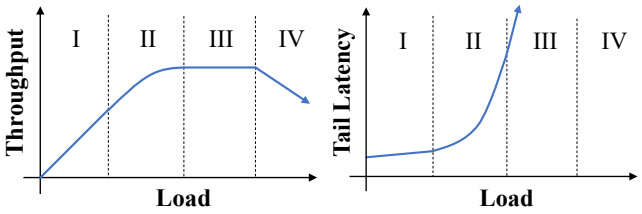
Unpredictabile blocking synchronization. If we could know which blocking synchronization a request will require to produce a response, an overload control simply check whether sum of the queueing delay of the required blocking synchronizations will violate its SLO. However, like per-bucket locks in the hash table, the blocking synchronizations a request will require is unpredictable, which makes it much harder to devise a useful congestion signal for overload control.

Low latency vs. high throughput. As exemplified with exisitng congestion signal, it is very challenging to maintain a balance between low latency and high throughput, especially with unpredictable mutex contention. Focusing on a latency-centric metric (e.g., end-to-end latency) easily under-utilize the server whereas focusing on a throughput-centric metric (e.g., request execution delay, CPU usage) leads to high tail latency.

Providing a way to abort the request in the middle of execution. Because of the unpredictability, an effort to admit incoming load to saturate the server is easily followed by high tail latency. To ensure each request meets its SLO, the RPC server should be able to drop the request in the middle of execution if it is likely to violate its SLO. However, partially executed request cannot be simply dropped because it might have allocated memory, modified the internal state, etc. Providing a way to revert intermediate state when dropping the request is not a trivial task.

3 System Design

We present Hoover, an overload control system with latency-aware locking synchronization. Figure 3 illustrates an overview of Hoover operation in an RPC server. Hoover has two main components: throughput-based credit management and latency-aware blocking synchronization. Hoover uses credit-based admission control where the server issues credits based on the server's overload status, and clients can send a request only when they possess a credit. A credit manager is responsible for adjusting the number of credits in the credit pool. In Hoover, it adjusts the number of credits based on the measured throughput to ensure that issued credit contributes to the actual throughput. With latency-aware blocking



(a) Throughput (b) Tail Latency

Figure 4: The performance of application with unpredictable lock contention without overload control

synchronization, when a request needs a lock, it first checks whether waiting for the lock will violate its SLO. If so, the request is dropped and returned to the clients with a rejection message. Throughput-based credit management and latency-aware blocking synchronization enable overload control even when the blocking synchronization is the bottleneck.

For the rest of this section, we first propose a simple way to control the overload of blocking synchronization with spinlocks with a certain condition. To devise an overload control for more general circumstances, we analyze the behavior with blocking synchronization contention and introduce how we design Hoover based on that.

3.1 Transfer Congestion of Blocking Synchronization to CPU

One simple solution to address the congestion of blocking synchronization could be to replace a blocking synchronization call with a non-blocking synchronization, e.g. to replace a mutex with a spinlock. This will transfer the blocking synchronization queue into the thread queue for CPU, which can be effectively handled by traditional congestion signals like CPU usage, latency, or request execution delay. However, this approach does not work all the time.

If the ratio of the lock time (the time in the critical section) to the overall request service time is large, most of the cores will end up waiting for the lock, hurting throughput and wasting CPU cycles. On the other hand, if the ratio of lock time to the request service time is small, other cores could do useful work outside the critical section, resulting in more efficient use of CPU and high throughput. For example, the application in Figure 1 cannot benefit from replacing mutex into spinlock because the ratio of lock time to the overall service time in slow path is around 1.0.

3.2 Throughput / Latency behavior with Unpredictable Blocking Synchronization

First, we describe how an application with unpredictable lock contention behaves with different load in general. We divide the application behavior into four different phases, where each phase has its own characteristics. Figure 4 illustrates how throughput and tail latency changes as load increases. Here we are considering a more challenging case the locks become the bottleneck before the CPU.

Phase I is the uncongested phase where none of the locks or CPUs are congested. In this phase, as the load increases, throughput grows linearly, and tail latency tends to increase marginally because the "bubble" in the queue caused by dynamics of request arrival that is frequently modeled with the Poisson process.

Phase II is the partially congested phase where a subset of locks are contended. As load increases, throughput increases sub-linearly because the requests waiting for a contending lock do not contribute to the throughput, resulting in a concave throughput line. Based on the application's characteristics, each application has a different concave line. Because the queues for contended locks keep growing, the tail latency grows exponentially.

Phase III is the congested phase where the additional incoming load does not contribute to the throughput because of the lock contention. Note that in Phase III, the CPU is not fully utilized yet. With increased load, the server will accept more requests because there are available CPU resources to handle incoming requests, but those excessive requests will end up waiting for a lock in the application. As a result, the throughput does not grow while tail latency increases because of increased queueing delay.

Phase IV is the congestion collapse phase where the CPU is also congested in addition to the locks. When CPU is congested without any overload control, throughput starts to degrade because of the livelock where some CPU cycles are used for processing incoming requests instead of doing useful work to process the requests. Tail latency still keeps growing since the requests additionally suffer from large thread queueing delays before getting a CPU for execution.

An ideal overload control should keep the server at some point in phase II where it can achieve high throughput (right side of phase II) or low tail latency (left side of phase II). Our system design starts from the question of whether we could achieve both high throughput and tail latency in phase II.

3.3 Performance-driven Credit-based admission control for throughput

Credit-based admission control provides a great way to bound the incoming load with bounded tail latency [7] by limiting incast size. With credit-based admission control, the problem becomes how much load the server should admit. If the load is bounded to the leftmost point of Phase II, it will provide low tail latency without request drop, and the best resource efficiency because all the CPU cycles are used for processing request, but it is more likely to under-utilize the server. On the other hand, if the load is bounded to the rightmost point of Phase II, it will provide the best possible throughput, but it leads to either high tail latency due to the queueing or less resource efficiency if it drops the requests.

Because each application has different trade-offs between throughput and latency, Hoover provides a way to specify the operating point with an *efficiency threshold* which ranges between 0 and 1. With the efficiency threshold of 1, Hoover

Algorithm 1 Performance-driven credit management

t_e: efficiency threshold
t_d: maximum drop threshold
C_{avail}: the size of credit pool
in_{last}: # of incoming requests in last micro-experiment
in_{cur}: # of incoming requests in current micro-experiment
out_{last}: # of outgoing responses in last micro-experiment
out_{cur}: # of outgoing responses in current micro-experiment
$drop_{cur}$: # of request drops in current micro-experiment
α: aggressiveness of step size
s: step size
n_c: the number of connections

repeat Every 4 * RTT
  $s \leftarrow \alpha \cdot n_c$
  if $drop_{cur} > t_d \cdot in_{cur}$ **then**
    $C_{avail} \leftarrow C_{avail} - s$
  else if $in_{cur} > in_{last}$ **then**
    // credit pool has increased in last micro-experiment
    if $(out_{cur} - out_{last}) > t_e \cdot (in_{cur} - in_{last}))$ **then**
      $C_{avail} \leftarrow C_{avail} + s$
    else
      $C_{avail} \leftarrow C_{avail} - s$
    end if
  else
    // credit pool has decreased in last micro-experiment
    if $(out_{last} - out_{cur}) > t_e \cdot (in_{last} - in_{cur}))$ **then**
      $C_{avail} \leftarrow C_{avail} + s$
    else
      $C_{avail} \leftarrow C_{avail} - s$
    end if
  end if
  $in_{last} \leftarrow in_{cur}$
  $out_{last} \leftarrow out_{cur}$
until Application exits

operates the system at the leftmost point of phase II in an effort to provide 100% resource efficiency. With the efficiency threshold of 0, Hoover operates the system at the rightmost point of phase II in an effort to increase the throughput regardless of the resource efficiency. With the efficiency threshold of 0.5, Hoover tried to operate at the point in phase II where increased load contributes at least 50% to the throughput, i.e., the point with tangential slope is 0.5 in the throughput graph in Figure 4 (a).

Inspired by BBR [6] and PCC [9] network congestion control, Hoover adjusts the credit pool size to operate the server at the target point with the desired efficiency threshold. For every 4 end-to-end RTTs including network RTT and request execution latency, Hoover conducts a micro-experiment with different global credit pool size, and it collects the data of the number of the incoming requests (in_{cur}) and the number of outgoing responses (out_{cur}) for each micro-experiment. Because it takes one RTT for a changed credit pool size to take an effect on throughput, Hoover does not collect the data for the first RTT of the micro-experiment. When the micro-experiment finishes, it computes the tangential slope of the current operating point with in_{cur} and out_{cur} together with the data in the previous micro-experiment and decides whether to increase the credit pool size or decrease. Hoover increases the credit pool size the same way as the original Breakwater to keep the benefit with a large number of clients. When it decides to decrease the credit pool size, the credit pool size is decreased with the same amount of the increase so that the credit management algorithm does not bias toward increment or decrement, assuming that the application does not share the resources with other applications.

A service operator may want to maintain the drop rate of the RPC server under a certain level even with performance degradation because the drop rate immediately impacts the expected number of retries until a request successfully receives a response. To cope with such a scenario, Hoover has a configurable *maximum drop threshold* parameter. Within a micro-experiment, if the drop rate exceeds the maximum drop threshold, the credit pool size is reduced regardless of the marginal throughput improvement. The full algorithm of credit management is described in Algorithm 1.

3.4 AQM with latency-aware synchronization for latency

With performance-driven credit-based admission control, the system is provided enough load to achieve target efficiency. However, if the efficiency threshold is smaller than 100%, the tail latency will be high because of the queueing delay in the congested data path. In order to push down tail latency so that most of the requests meet the SLO, we need a way to drop the request in the middle of the data path. Dropped requests then are returned to the clients instead of waiting for contending resources, and clients can retry the request to another possibly uncongested server.

With our observations that the end-to-end latency may be dominated by the blocking synchronization queueing delay (e.g., mutexes and conditional variables), we devise latency-aware blocking synchronization APIs. By telling the queueing delays of the waiter queue, they enable request drop if the request is expected to violate its SLO because of the long queueing delay.

```
bool mutex_lock_if_uncongested(mutex_t *);
bool condvar_wait_if_uncongested(condvar_t *,
                                 mutex_t *);
```

For each application, the "queueing budget" is provided by service operators based on server-level SLO, service time distribution, and network conditions. Queueing budget represents the maximum queueing delay a request can tolerate to meet its SLO. In the server, queues for different types of resources (thread queue for CPU, mutex waiter queue for mutex, conditional variable waiter queue for conditional variable)

keep track of its queueing delay, and each request does accounting on accumulated queueing delay it has experienced. When a request waits for a mutex or a conditional variable with a latency-aware synchronization call, Hoover determines whether the request will violate its SLO if it waits for the resource. More specifically, it checks whether the sum of accumulated queueing delay of the request, the queueing delay of the resource, and thread queueing delay (because once it grabs the resource after the queueing delay, it needs to wait for CPU to execute further) exceeds the queueing budget. If so, the API returns false immediately without waiting for the resource; otherwise, it waits for the resource and returns true once it grabs the resource. When the request needs to be dropped with the return value of false, developers are responsible for writing a clean-up code that frees allocated memory, releases currently holding locks, reverts intermediate operations and state changes, etc.

Hoover uses accumulated queueing delay instead of sojourn time because it's hard to know the progress on request processing at the time of determining the likelihood of SLO violation. For example, with sojourn time to decide whether a request will violate its SLO, i.e., checking whether the sum of sojourn time and the mutex queueing delay exceeds SLO, the mutex will accept more requests with a small progress, and the lock will suffer more congestion. To avoid the necessity of tracking a request's progress to its completion, we separate processing time and queueing delay and only use queueing delay to determine whether the request will violate its SLO.

4 Implementation

Hoover requires CPU-efficient operating system so that it can provide the best possible goodput with minimal overhead. As an operating system, we use Shenango [18], an CPU-efficient operating system designed for latency-sensitive workload with lightweight user-level threads and fast core allocations. We implement queueing delay measurement in mutex and conditional variable, latency-aware synchronization abstraction, and per-request queueing delay accounting feature in Shenango's runtime library. For an overload control implementation, we use Breakwater, an state-of-the-art overload control system for µs-scale RPCs. With credit-based admission control and SLO-aware latency-based AQM, Breakwater provides high throughput and low tail latency even with large number of clients. We modify credit management logic so that it adjusts the credit pool size based on the measured throughput changes and AQM logic so that it supports in-application request drop during execution.

Measuring queueing delay. We instrumented the mutex queue and conditional variable queue in the same way as Breakwater instrumented packet queue and thread queue for delay measurement [7]. Whenever a new request is enqueued to the waiter queue of a mutex or a conditional variable, Hoover timestamps the request. When the queueing delay is requested, it returns the difference between current time stamp and the timestamp of the oldest request in the queue. Because Shenango provides a way to timestamp requests with small overhead, and reading queueing delay requires only one timestamp read and subtract operation, Hoover can measure queueing delay for mutex or conditional variable without much overhead.

Per-request queueing delay accounting To determine whether a request is likely to violate its SLO, Hoover needs to keep track of accumulated queueing delay of each request. Whenever a new request is enqueued to a thread queue, mutex waiter queue, or conditional variable waiter queue, Hoover timestamps the enqueue time (Hoover already does this for measuing queueing delay!). When a request finished waiting in a queue and dequeued, Hoover adds the difference of current timestamp and enqueue timestamp to the accumulated queueing delay.

Modifying application for Hoover Since a developer should handle request drop with lock contention to clean-up the intermediate state, Hoover requires modification in the application code. With no pre-allocated memory or internal state change during the request execution like an example application in Figure 1, the developer can simply specify current request is dropped and return like following:

```
void srpc_request_handler() {
    ...
    // slow path
    if (!mutex_lock_if_uncongested(&lock)) {
        get_rpc_ctx()->drop = true;
        return;
    }
    work();
    mutex_unlock(&lock);
    ...
}
```

5 Evaluation

Our evaluations answers the following key questions:

1. Can Hoover prevent overload while maximizing goodput under lock contention for a wide range of applications?
2. How do Hoover's mechanisms contribute to its performance?
3. How does Hoover perform without application modifications?
4. How is Hoover's performance sensitive to the tuning of its parameters?

5.1 Evaluation Setup

Testbed: We use 11 xl170 nodes in the Cloudlab Utah cluster. Each node has a ten-core (20 hyper-threads) Intel E5-2640v4 2.4GHz CPU, 64GB ECC RAM, and a Mellanox Connect X-4 25Gbps NIC. Nodes are connected throughput a single Mellanox 2410 25Gbps switch. The network RTT between any two pair of nodes is 10 µs. We use one node as an RPC server and the other ten nodes as RPC clients. The server application uses up to 10 hyper-threads with Shenango's dynamic

core allocation for processing requests, and the client application uses dedicated 16 spinning hyper-threads to generate the load. All nodes run Shenango's IOKernel in a dedicated hyper-thread in addition to the application.

Baseline: We compare Hoover to Breakwater, a state-of-the-art overload control system for micro-second scale RPCs supporting a large number of clients with sporadic demands. Breakwater exploits two key techniques for efficient overload control: Credit-based admission control with demand speculation and request execution delay-based AQM. With credit-based admission control, a Breakwater RPC server issues credits based on its request execution delay signal, and clients can send a request only with credits. By scheduling incoming load with credits in request granularity, Breakwater limits the size of the incast. To reduce the overhead of synchronizing demand information between the server and clients, the RPC server speculates the clients' demand and overcommits the credits. An incast caused by overcommited credits are handled by AQM which drops the request if it is expected to violate the SLO.

Server-level SLO: We set the server-level SLO to 10× the sum of average network RTT and the average service time with no congestion. The way we set the SLO is inspired by Breakwater [7] and other recent works [8, 19].

Evaluation metrics: To incorporate throughput, latency, and SLO into one metric, we compute goodput that is the throughput of the requests whose latency is below SLO. A high goodput can only be achieved with high throughput and lower latency than SLO. For some experiments, we report server-side average CPU usage and drop rate. For drop rate, we compute the ratio of dropped requests to the total number of requests received at the server during an experiment. We run the experiments for 8 seconds and collected the data for the last 4 seconds to capture the steady-state behavior if not specified otherwise.

Parameter settings: For Breakwater, we use the default setting for $\alpha = 0.1\%$ and $\beta = 2\%$ and recommended parameter for target delay and AQM threshold [7]. For example, with exponential service time distribution of 10 µs average with 200 µs SLO, we set target delay to 80 µs and AQM drop threshold to 160 µs. For Hoover, we use an efficiency threshold (e_t) of 25% by default (for example, for the synthetic application with one unpredictable mutex) but tune to produce the best goodput with real-world applications. As each application has its own concave throughput line in phase II in Figure 4 (a), different application achieves the best goodput with different efficiency threshold. We compute the queueing budget by deducting 99th percentile network delay and 99th percentile request service time with no congestion from SLO. For all experiments, we set the maximum drop threshold to 100% to maximize the goodput, but the drop rate doesn't exceeds the 15% for all the experiments we conduct with a wide range of applications.

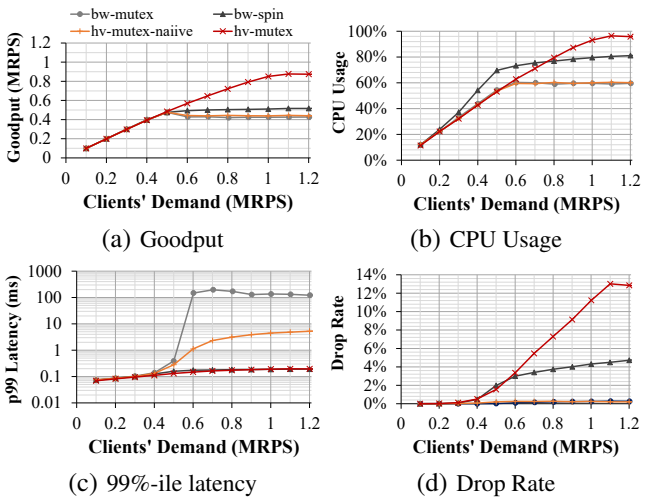


Figure 5: Performance of Breakwater (bw-) and Hoover (hv-) with example application of Figure 1 ($p = 20\%$) with mutex (-mutex) and spinlock (-spin) implementation

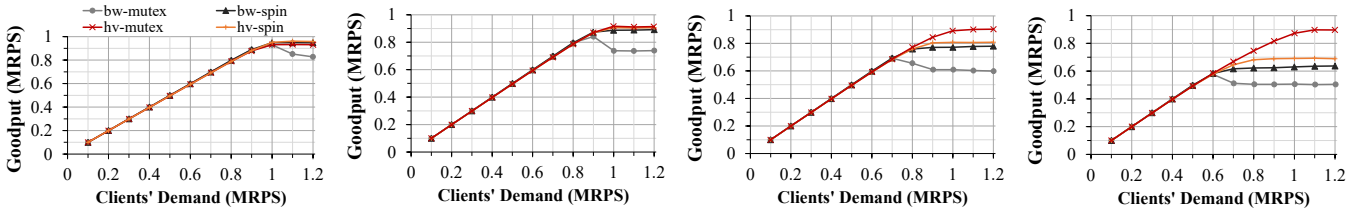
5.2 Application with Unpredictable Mutex Contention

First, to demonstrate Hoover's ability to handle unpredictable mutex contention, we repeat the experiment in Section 2.1 with $p = 20\%$. With a network RTT of 10 µs and the average service time of 10 µs, we set the server-level SLO to 200 µs.

Figure 5 shows goodput, CPU usage, 99th percentile latency, and drop rate of original Breakwater, Breakwater with spinlock, and Hoover. With $p = 20\%$ and slow path's capacity of 100k reqs/s, larger than 500k reqs/s of the clients' demands saturate the slow path. As we discussed in Section 3.1, even though spinlock marginally improves the goodput because Breakwater effectively bounds the tail latency, its goodput is still far from the best. Because request execution delay signal Breakwater uses can capture the lock contention, it can handle the overload with credit-based admission control and request drops. However, its CPU usage is less efficient than with mutex, i.e., it requires more CPU cycles per request on average, and more critically, it suffers from head-of-line-blocking. As the slow path has a high ratio of the lock time to the overall service time and a single mutex contended by a large number of the requests, all the cores end up waiting for the global mutex, which leads to low goodput even with high CPU utilization.

Using Hoover without modifying the application code (hv-mutex-naiive) shows similar performance in terms of goodput, CPU usage, and drop rate. Without application modification, Hoover controls the overload only by adjusting the credit pool size based on throughput measurement without any request drop. With $p = 20\%$, as Hoover can observe 80% of throughput increase due to the fast path, it increases the credit pool size. With increased credit pool size, the mutex eventually will be congested. Because of the increased average RTT[3] due to the long mutex queue, the increment speed of the credit pool

[3]Note that Hoover uses end-to-end RTT for credit management opposed to Breakwater that updates credit pool size every network RTT



(a) $r = 0.2$ (b) $r = 0.4$ (c) $r = 0.6$ (d) $r = 0.8$

Figure 6: Effect of lock time ratio (r) on performance of Breakwater (bw-) and Hoover (hv-) for example application of Figure 1 ($p = 20\%$) with mutex (-mv) and spinlock (-spin) implementation.

will be reduced as the queueing delay grows larger and larger. When the credit pool size reaches its maximum, it will not observe increased throughput by fast path anymore, starting to reduce the credit pool size. However, Hoover would not observe lower throughput with reduced credit pool size because the mutex keeps draining its waiter queue, producing output. As a result, the credit size keeps reducing until it eventually observes the throughput drop after draining all requests in the mutex waiter queue. Because of slower credit pool increment speed and credit pool size reducing logic, Hoover achieves lower tail latency.

After replacing traditional mutex lock call with Hoover's latency-aware API, which enables request drop ahead of the mutex, Hoover achieves up to 2.06× of goodput compared to the Breakwater with mutex implementation. Hoover is able to achieve such a high goodput by admitting more requests with throughput-based credit management while it drops requests in the slow path if they are likely to violate its SLO. Because it utilizes the fast path more and keeps mutex queueing delay small with request drop, it achieves higher goodput with a high CPU utilization and a high drop rate.

5.3 Effect of Lock Time Ratio on Performance

Lock time ratio (the ratio of the time in the critical section to the overall service time) impacts the performance of Breakwater and Hoover. To better understand the application with different lock time ratios and to demonstrate that Hoover can consistently provide the benefits, we conduct experiments with example application in Figure 1 with $p = 20\%$. To set a different lock time ratio of slow data path (r), we modified the application so that the request does $(1-r)$ $(0 \leq r \leq 1)$ of the work before it waits for the mutex. Once the request grabs the mutex it completes r of the work while holding the mutex. For example, with $r = 0.2$, 80% of the work is done outside the critical section while the other 20% is done with holding the mutex.

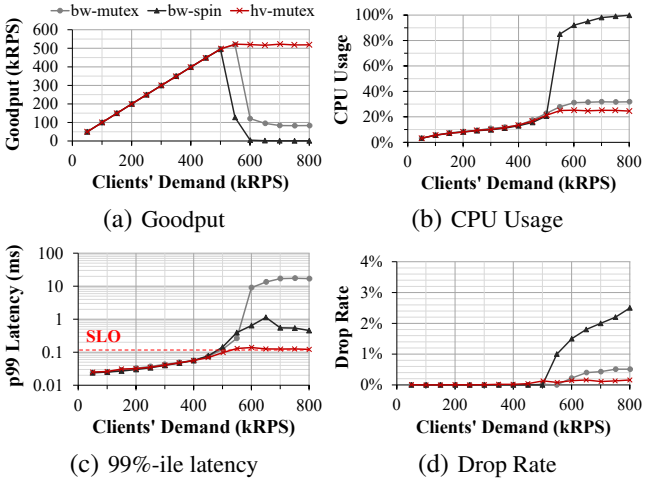
Figure 6 shows the goodput of Breakwater and Hoover with mutex and spinlock implementation with $r = 0.2$, 0.4, 0.6, and 0.8. When r is small, the mutex will experience less congestion with its high throughput. As a result, mutex queueing delay will not grow severely, making Breakwater and Hoover produce similar goodput regardless of the lock implementation. As r becomes higher, Breakwater suffers from high mutex queueing delay and head-of-line-blocking where the fast path is under-utilized because of a large number of request waiting for the mutex in the slow path. As a result, Breakwater experiences severe goodput degradation with higher r. Replacing mutex into spinlock with Breakwater (bw-spin) could achieve near-best goodput when r is lower because the amount of time busy-spinning the core waiting for the mutex is small. However, it becomes less efficient with higher r with wasting CPU. With the example application in Figure 1, spinlock with Breakwater can achieve better goodput regardless of r as it makes admission control works by moving mutex congestion into CPU congestion.

Hoover achieves better goodput than Breakwater regardless of r and the lock implementation, providing the benefits for a wide range of applications. Hoover with spinlock implementation achieves near-best performance with small r, but it gets degraded with higher r with more CPU waste with busy waiting. Hoover with mutex implementation achieves near-the-best performance in all scenarios with different r thanks to throughput-based credit management and request drop in the middle of data path with latency-aware blocking synchronization.

5.4 Application with High Lock Time Ratio

To verify the advantage of Hoover for a wide range of applications with different characteristics, we first consider Memcached [2] that has a relatively long lock time compared to the overall service time.

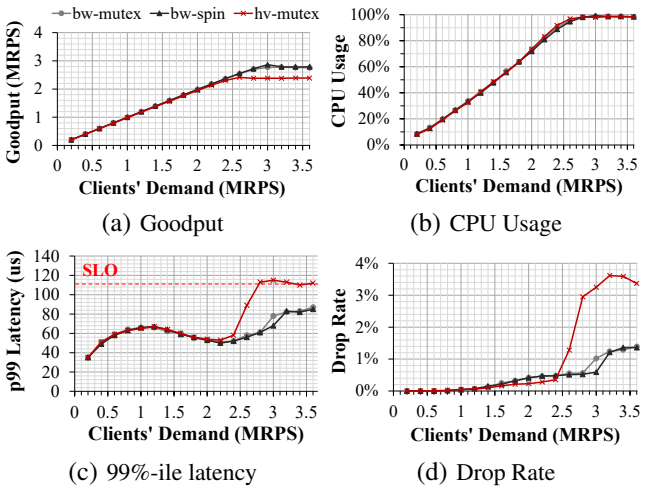
Application Structure and Characteristics: Memcached is a popular in-memory storage application for small key-value pairs. The key-value pairs are stored in a giant hash table with a per-bucket lock called `item_locks[]` that is implemented in mutex. There are $2^{\texttt{hash\_power}}$ number of buckets in the hash table. For a request to modify, add, delete, or retrieve the key-value pairs, it requires a corresponding per-bucket lock. Memcached manages memory with pages (1MB by default) and slab classes. Each slab class is responsible for the different sizes of the item. To allocate and free the memory in Memcahced, the global lock called `slabs_lock` is required to access a global table responsible for tracking allocated and freed pages for each slab class. When Memcached server receives a GET request, it reads the value in the hash table for the key in the request while holding corresponding `item_locks[]` and returns the value to the client. For a SET request, Memcached server first allocates a chunk in a



(a) Goodput (b) CPU Usage

(c) 99%-ile latency (d) Drop Rate

Figure 7: Memcached performance of Breakwater (bw-) and Hoover (hv-) with mutex (-mutex) and spinlock (-spin) implementation for VAR workload



(a) Goodput (b) CPU Usage

(c) 99%-ile latency (d) Drop Rate

Figure 8: Memcached performance of Breakwater (bw-) and Hoover (hv-) with mutex (-mutex) and spinlock (-spin) implementation for USR workload

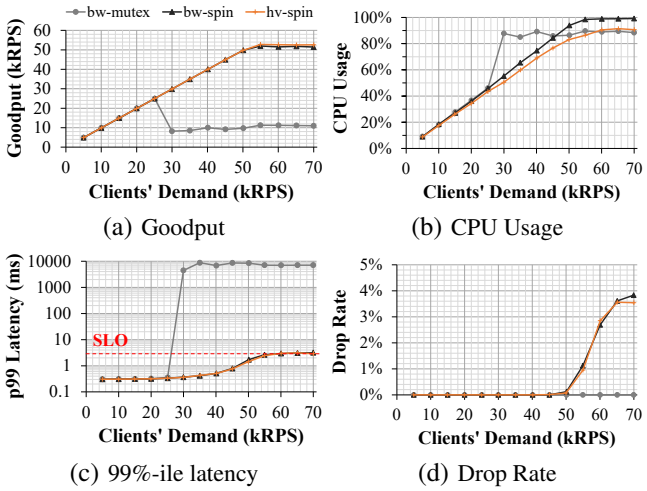
slab class for the value size to store the value, which requires `slabs_lock`. After copying the value from the request to the allocated chunk, it updates the hash table with `item_locks[]` and frees the old chunk (or allocated chunk if store operation was unsuccessful) with `slabs_lock`. In Memcached, both `slabs_lock` and `item_locks[]` can be congested. When there are a small number of buckets compared to the number of key-value pairs or a large number of requests need access to the same bucket in the hash table, `item_locks[]` will be congested. With a high volume of SET requests, `slabs_lock` can be congested.

Application Modification: To support request drop during the request execution, we replaced some of the existing mutex lock calls with Hoover's latency-aware lock API. First, we drop the request if waiting for `item_locks[]` will violate the request's SLO. By marking SET or GET operations as a failure, Memcached handles the clean-up (e.g., freeing the allocated chunk) for dropped requests. Second, we drop the request if waiting for `slabs_lock` will violate the SLO for SET requests' chunk allocation. Because chunk allocation is a very first operation before proceeding further, no clean-up is necessary in this case. Note that we do not drop the request when it frees a chunk even though `slabs_lock` is congested as it is an essential operation for memory management. For some applications, it might not make sense to drop SET requests. In that case, a developer can simply use a traditional mutex lock call instead of Hoover's latency-aware one for SET-related operations to avoid the drops. For spinlock implementation, we convert `slabs_lock` and `item_locks[]` to spinlocks.

Workload and Configuration: For Memcached experiments, we use two workloads with the highest volume from Facebook [22]: VAR and USR. VAR is the workload for server-side browser information. VAR is a SET-heavy workload where 82% of the requests are SET requests, and the other 18% of the requests are GET requests. The key distribution of the VAR workload is skewed. 10% of the keys are used by 90% of the requests. USR is the workload for user-account status information. USR is a GET-heavy workload where 99.8% of the requests are GET requests, and 0.2% of the requests are SET requests. In the USR workload, about 20% of the keys are used by 80% of the requests. We approximately follow the key and value size distribution for each workload as described in the paper [22]. We use 100,000 key-value pairs and use the hash power of 17, providing 131,072 buckets in the hash table to avoid the hash collision. 1,000 clients are generating requests with an open-loop Poisson process. Because a SET request takes 10 µs and a GET request takes less than 1 µs on average without congestion, SLO is set to 110 µs for USR workload and 200 µs for VAR workload. With a parameter tuning, we set the efficiency threshold (e_t) to 1% that produces the highest goodput for Hoover.

Figure 7 shows the performance of Breakwater and Hoover for VAR workload. Because of the SET-heavy characteristics of VAR workload, `slabs_lock` easily becomes the bottleneck by a large number of SET requests. With a single bottleneck of `slabs_lock`, Breakwater's goodput collapses when the server is overloaded with clients' demand exceeding 500k reqs/s. With a single bottleneck, replacing mutex with spinlock doesn't fix the problem, and all the cores end up waiting for a single lock, causing a head-of-line-blocking problem. With a head-of-lien blocking, the throughput is significantly degraded. In addition, because all the cores except one are waiting for a single lock with small throughput, it forms a long enough queue, and almost all requests will violate its SLO. As a result, it achieves nearly zero goodput while wasting a lot of CPU cycles.

On the other hand, Hoover efficiently controls the overload with a single bottleneck, achieving up to 6.2× goodput and 140× lower tail latency with small CPU utiliza-



(a) Goodput (b) CPU Usage

(c) 99%-ile latency (d) Drop Rate

Figure 9: Breakwater's performance of Lucene (bw) with mutex (-mutex) and spinlock (-spin) implementation and Hoover's performance of Lucene with spinlock implementation without application modification

tion. Because 82% of the requests follow a slow path with a high SET request rate, throughput-based credit management won't increase the size of the credit pool aggressively after `slabs_lock` is congested, leading to a small request drop rate.

Figure 8 shows the performance of Breakwater and Hoover for USR workload. Because of the GET-heavy characteristics of USR workload and a large number of buckets in the hash table compared to the number of key-value pairs, the CPU becomes the bottleneck before any mutex. As a result, Breakwater with both mutex and spinlock implementation demonstrates similar performance. However, Hoover achieves less goodput, higher tail latency, and higher drop rate than Breakwater. This experiment reveals a limitation of Hoover. Hoover adjusts its credit pool size every four end-to-end RTTs because it needs to collect useful enough data on throughput to make an adjustment, whereas Breakwater adjusts its credit pool size for every one network RTT. Due to the slow responsiveness of Hoover, it has a weak point of micro-congestion with a small request service time that requires fast responsiveness. As a result, Hoover achieves up to 15% less goodput than Breakwater.

5.5 Application with Low Lock Time Ratio

In this section, we consider another real-world application, Lucene [1] that has a relatively long lock time compared to the overall request service time.

Application Structure and Characteristics: Lucene [1] is a search library widely used in the industry. When a document is added to the Lucene, it constructs the table called "segment" which contains inverted indices which map from a word to document IDs. With the default configuration, Lucene creates a segment for each document and merges multiple segments into one for every 10 documents. Further, when the number of the segments for 10 documents reaches 10, Lucene merges them into one single segment for 100 documents, etc. Therefore, each segment contains inverted information for 10^n number of documents (n is the number of merge operations). When a Lucene receives a search request, it first retrieves the list of document ID by binary searching the segments, which requires segment lock while searching. Once it has the list of document IDs, Lucene scores each document based on a pre-defined metric (e.g. the number of the appearance of the words in the document). The score is pre-calculated, but the request requires the per-document lock to get the score. After getting the scores of all the documents, it sorts them based on the score and returns the result to the client. There are two main types of locks required during a search: segment lock and document lock implemented with mutex in the original code base. Because segments and documents are the shared resources, segments locks and documents locks, especially for popular documents, could be contended with a large number of requests.

Application Modification: As Lucene is more complex than Memcached with lock hierarchy with nested locks and more memory allocation/deallocation to store intermediate results, we explore the option of applying Hoover without application modification. For spinlock implementation, we change the object lock used for segment lock and document lock to spinlock.

Workload and Configuration: We port the Lucene library implemented in C++ [20] to Shenango. We populate the Lucene application with a realistic dataset of 403,619 COVID-19 related tweets [5] posted between 27th and 28th November 2021. 1,000 RPC clients are issuing requests for single-term search queries with the open-loop Poisson process. The search term is sampled from the word distribution in the data set, i.e., the more frequently a word appeared in the tweets, the more likely to be selected as a search query. We exclude stop words (e.g., "a", "the", "and", etc.) when constructing word distribution. All the tweets are loaded into the RAM with `RAMDirectory` before the experiment, and the tweets are not deleted or modified during an experiment. Because the request takes 270 µs on average when the server is not congested, we set the SLO to 2.8 ms considering 10 µs of network RTT. With a parameter tuning, we set the efficiency threshold (e_t) to 1% that produces the highest goodput.

Figure 9 demonstrates the performance of Breakwater and Hoover with mutex and spinlock implementation. Breakwater with mutex implementation exposes the problem with unpredictable blocking synchronization in Lucene with low goodput due to extremely high tail latency. When one of the mutexes starts to be congested (from clients' demand of 30k reqs/s), its CPU usage grows super-linearly because, unlike the example application with a single mutex, the requests consume CPU cycles before being stuck into one of the mutex queues. As clients' demand increases further, CPU usage flattens out around 90% since the credit pool size reaches its maximum and most requests are waiting for a mutex.

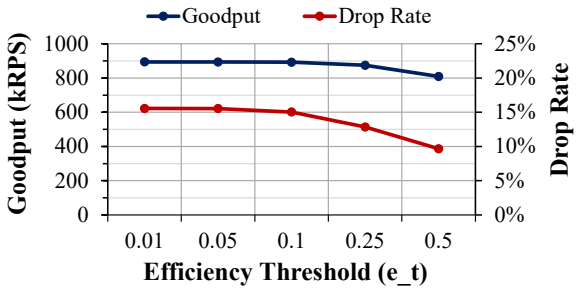


Figure 10: Hoover parameter sensitivity (efficiency threshold, e_t)

For Lucene, just replacing mutexes with spinlocks with Breakwater improves the performance significantly, achieving up to 4.6× of the goodput by fully utilizing the CPU and 2,244× lower tail latency with request drops. Transferring lock contention into CPU congestion by replacing mutexes with spinlocks is effective for Lucene because the lock time is small compared to the overall service time. While its average service time is 270 µs, Lucene has only one binary search and list look-up with a segment lock, and one hash table look-up with document lock.

Even for the case where the spinlock fixes the problem with unpredictable blocking synchronization, Hoover improves performance and CPU efficiency further without application modification. hv-spin line in Figure 9 shows the performance of Hoover with spinlock implementation of Lucene without request drops in the middle of request execution. Because all the request drops are due to CPU congestion, Breakwater and Hoover show a similar drop rate. However, Hoover still provides its benefits with throughput-based credit management, leading to slightly better (up to 2.2% higher) goodput and better CPU efficiency with less CPU Usage (10% less compared to Breakwater). This is due to Hoover's performance-driven decision-making. Hoover increases its credit pool size if it observes the throughput increase in the micro experiment even with CPU congestion and decreases its credit pool size as long as it observes no sufficient throughput drop because of lock contention even though there are idle cores available.

5.6 Parameter Sensitivity

Hoover has efficiency threshold parameter that is required to be tuned to make it behave in the desired way. Small efficiency threshold will admit more requests by issuing credits more aggressively, resulting in a high drop rate while a large efficiency threshold will issue a credit more conservatively. To study the trade-off between goodput and drop rate with different efficiency thresholds, we repeat the experiment with an example application in Figure 1 with $p = 20\%$. Figure 10 shows the goodput and drop rate of Hoover with different efficiency thresholds when the clients' load is 1.2M reqs/s (1.3× capacity). The default value of 25% we used provides the near-highest throughput with a reasonable drop rate of 13%, but reducing the threshold further improves the goodput with a higher drop rate. $e_t = 0.01$ achieves 895k reqs/s with drop rate of 15.5% whereas $e_t = 0.5$ achieves 810k reqs/s (10% less compared to $e_t = 0.01$) with drop rate of 9.7%.

6 Discussion

Service-level SLO and server-level SLO. In this paper, we consider a single server and use server-level SLO for configuration. Server-level SLO can be derived from service-level SLO with the expected number of retries. For example, if a server has a maximum drop rate of 20%, it requires three retries to achieve 99% success rate ($1 - 0.2^3 = 99.2\%$). With the expected number of retries, the service-level SLO can be evenly distributed to each try to the server for server-level SLO.

Generalizing latency-aware request drop. The idea of Hoover to drop the request if it is expected to violate the SLO with queueing delay measurement and per-request queueing delay tracking can be generalized to other types of resources. For example, a system requiring conditional disk access (e.g. with caching), can drop the request if the I/O submission queue delay will make the request violate its SLO.

Expanding overload signal to multi-hierarchy microservices. [11] observed a challenging overload scenario in multi-hierarchy microservices where the tail latency of upstream service (NGINX) spikes more than 10× while its CPU usage remains low due to the blocking network socket with HTTP1. Because long queueing delay of blocking network socket cannot be detected with the overload signal they use (CPU Usage), it does not trigger auto-scaler to launch a new instance causing high tail latency. The overload signal with throughput gain Hoover use can handle such an overload case providing low tail latency together with latency-ware blocking APIs that enable request drop during the request execution. Further, we believe that Hoover's approach can be extended to multi-hierarchy microservices where upstream microservices control the incoming load with throughput-based credit management, and downstream microservices can drop the requests if it is bound to violate its SLO.

7 Related Work

Measurement-based network congestion control. BBR [6] and PCC [9] control network congestion based on the performance measurements during micro-experiments. BBR measures two metrics: minimum RTT and maximum bandwidth with multiple. The way it searches for the maximum bandwidth in the start-up phase resembles Hoover. It doubles its window size and looks whether the throughput has increased by at least 25% more than the previous micro-experiment. When the measured throughput gain is less than 25%, it stops doubling the window and finalizes the maximum bandwidth of the network. Based on the measurement, BBR sets its window size to BDP, the minimum RTT multiplied by the maximum bandwidth. In PCC, the system operator must define a utility function. For every round, after two micro-experiments with reduced window size and increased window size, PCC decides whether to increase or decrease the window size based

on which achieved higher utility with the micro-experiments. Hoover is similar in that it also performs micro-experiments, but with a new focus on overload control in the presence of lock contention.

Overload Control. To avoid congestion collapse where parsing incoming network packets, an overload control system tries to bound the incoming requests to prevent overload. Overload can be detected using several metrics. Breakwater [7] and DAGOR [23] use request execution time, the delay between request arrival and the request execution, and SEDA [21] and ORCA [14] use response time as a congestion signal. The way a system controls the overload also differs across these systems. Breakwater utilizes credit-based admission control with AQM, DAGOR utilizes priority-based admission control with AQM, SEDA controls the request sending rate at the client-side, and ORCA uses TCP-like window-based approach at the client-side.

Auto-scaling. Auto-scales [3, 12, 15] dynamically replicate the service when it is overloaded. While overload control handles the server contention in a smaller time scale, such as micro-burst during the operation, auto-scaling handles the overload in a longer time scale. If a server is consistently overloaded, launching an additional replica with auto-scaling is a more fundamental solution, but as replicating the machine takes time, overload control is still needed to accommodate transient states. The most common metric for auto-scaling is CPU usage. For example, Amazon AWS auto-scaler [3] replicates the service when it observes the average CPU utilization higher than 60%. A CPU usage-based auto-scaling cannot handle the blocking synchronization contention even in the longer time scale where CPU utilization remains low while latency spikes.

try_lock. Pthreads and other common threading packages provide a way to check whether a mutex is contended or not, called *try_lock*, by checking whether the mutex is currently held by another thread. With try_lock, a developer can drop requests in a similar way to Hoover, but try_lock is not appropriate for managing congestion: because try_lock only tells you whether the mutex queueing delay is zero or not, if a developer uses it to drop requests, it will achieve very short tail latency but suffer from low throughput.

8 Conclusion

In this paper, we present Hoover, an overload control system that handles blocking synchronization contention with throughput-based credit management and latency-aware blocking synchronization. Hoover achieves high goodput and good CPU efficiency for a wide range of applications. Hoover adjusts the size of the credit pool based on the measured throughput gain with micro-experiments, avoiding over-issuing credits when blocking synchronization is a bottleneck. To ensure each request does not violate its SLO, Hoover provides a way to drop the request in the middle of request execution with latency-aware blocking synchronization. Before Hoover enqueues a request to the waiter queue, it checks whether waiting for the lock will violate the SLO and drops it if it is expected to. Our evaluation of Hoover demonstrates that it outperforms state-of-the-art overload control systems in most cases. In particular, Hoover achieves 6.2× higher goodput and 140× lower 99th percentile latency with Memcached for realistic workload than Breakwater.

References

[1] Apache Lucene. http://lucene.apache.org/.

[2] Memcached. http://memcached.org/.

[3] AWS Auto Scaling, 2020. https://aws.amazon.com/autoscaling/.

[4] HAProxy, 2020. http://www.haproxy.org/.

[5] J. M. Banda, R. Tekumalla, G. Wang, J. Yu, T. Liu, Y. Ding, K. Artemova, E. Tutubalina, and G. Chowell. A large-scale COVID-19 Twitter chatter dataset for open scientific research - an international collaboration, May 2020. https://doi.org/10.5281/zenodo.3723939.

[6] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, 2016.

[7] I. Cho, A. Saeed, J. Fried, S. J. Park, M. Alizadeh, and A. Belay. Overload control for μs-scale rpcs with breakwater. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 299–314, 2020.

[8] A. Daglis, M. Sutherland, and B. Falsafi. Rpcvalet: Ni-driven tail-aware balancing of μs-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–48, 2019.

[9] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. {PCC}: Re-architecting congestion control for consistent high performance. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 395–408, 2015.

[10] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *International conference on World Wide Web*, 2004.

[11] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud &

edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[12] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Adaptive, model-driven autoscaling for cloud applications. In *11th International Conference on Autonomic Computing ({ICAC} 14)*, pages 57–64, 2014.

[13] M. Klein. Lyft's envoy: Experiences operating a large service mesh. 2017.

[14] B. C. Kuszmaul, M. Frigo, J. M. Paluska, and A. S. Sandler. Everyone loves file: File storage service (FSS) in oracle cloud infrastructure. In *ATC*, 2019.

[15] M. Mao, J. Li, and M. Humphrey. Cloud auto-scaling with deadline and budget constraints. In *2010 11th IEEE/ACM International Conference on Grid Computing*, pages 41–48. IEEE, 2010.

[16] J. C. Mogul and K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[17] NGINX Documentation: Limiting Access to Proxied HTTP Resources, 2020. https://docs.nginx.com/nginx/admin-guide/security-controls/controlling-access-proxied-http.

[18] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 361–378, 2019.

[19] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[20] B. van Klinken. Lucene++. https://github.com/luceneplusplus/LucenePlusPlus.

[21] M. Welsh and D. Culler. Overload management as a fundamental service design primitive. In *SIGOPS European Workshop*, 2002.

[22] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Characterizing facebook's memcached workload. *IEEE Internet Computing*, 18(2):41–49, 2013.

[23] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang. Overload control for scaling wechat microservices. In *SoCC*, 2018.