Mitigating Compute Congestion for Low Latency Datacenter RPCs

by

Inho Cho

B.S., Korea Advanced Institute of Science and Technology (2015)
M.S., Korea Advanced Institute of Science and Technology (2018)

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2023

© 2023 Inho Cho. All rights reserved.
The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Inho Cho
Department of Electrical Engineering and Computer Science
August 31, 2023

Certified by: Adam M. Belay
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by: Mohammad Alizadeh
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Mitigating Compute Congestion for Low Latency Datacenter RPCs

by

Inho Cho

Submitted to the Department of Electrical Engineering and Computer Science on August 31, 2023 in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Latency-sensitive applications in recent datacenter workloads, such as interactive machine learning inference, high-frequency algorithm trading, cloud gaming, and interactive AR/VR applications impose stringent latency requirements. These applications heavily rely on low-latency RPCs as an essential building block, often executed in mere microseconds through parallel computations and in-memory operations. Given the high fan-out RPC traffic patterns typical of these applications, it's imperative to minimize tail latency to maintain end-to-end latency within its service level objectives (SLO).

With the innovations in datacenter networks and the end of Dennard scaling, congestion is now moving from networks to compute resources. This thesis introduces two systems, Breakwater and LDB, designed to mitigate and diagnose compute congestion, each targeting different sources of tail latency. Breakwater aims to alleviate CPU congestion and lock contention during intermittent server overload, while LDB furnishes developers with a tool to diagnose the functions causing high tail latency with low overhead.

Thesis Supervisor: Adam M. Belay
Title: Professor of Electrical Engineering and Computer Science, MIT
Thesis Supervisor: Mohammad Alizadeh
Title: Professor of Electrical Engineering and Computer Science, MIT
Thesis Committee Member: M. Frans Kaashoek
Title: Professor of Electrical Engineering and Computer Science, MIT
Thesis Committee Member: Ahmed Saeed
Title: Professor of School of Computer Science, Georgia Tech

Acknowledgments

This thesis owes its completion to the assistance and support extended by numerous individuals. First and foremost, I would like to express my gratitude to my esteemed advisors, Adam Belay and Mohammad Alizadeh, for their invaluable guidance, patience, and unwavering support throughout this journey. Adam's remarkable insight and intuition in research have consistently inspired me. He has been generous in providing detailed feedback on various aspects, including implementation and how to better motivate my research projects. Mohammad, an exceptional educator and attentive listener, has always gone above and beyond. He is always willing to listen to my thoughts and ideas and has taught me the art of conveying complex ideas to the public in simpler words. His astute understanding of the field allows us to place our work in a broader context with clarity. Both my advisors possess the virtue of immense patience, readily accommodating even my minutest concerns. They have graciously invested their time in valuing my opinions and ideas, making me feel heard and supported throughout the research. Their guidance has been invaluable to my academic growth and development.

In addition to my advisors, I would like to acknowledge Frans Kaashoek and Ahmed Saeed for their contributions as members of my thesis committee. I appreciate the considerable time and effort they dedicated to guiding my dissertation work.

I also thank my major collaborators, Ahmed Saeed and Seo Jin Park, for their dedicated effort and invaluable contributions with extensive discussions and feedback. Their support and availability whenever I need them have been truly remarkable. Working alongside such exceptional mentors has been a great privilege that I deeply cherish.

Thank you to Josh Fried, Zain Zhenyuan Ruan, and Gohar Irfan Chaudhry for their support, discussion, and contributions to this work. I also thank other faculties and fellow graduate students in Parallel and Distributed Operating Systems (PDOS) and Network and Mobile Systems (NMS) group: Hari Balakrishnan, Manya Ghobadi, Frans Kaashoek, Nickolai Zeldovich, Robert Morris, Henry Corrigan-Gibbs, Lei, Pari, Arjun, Venkat, Vibhaa,

Akshay, Sanjit, Derek, Frank C., Anish, Jon, Amy, Jonathan, Tej, Sudarsanan, Pouya, Will, Yun-Sheng, Ariel, Upamanyu, Lily, Chenning, Alex, Arash, Zhizhen, Mingran, Pantea, Mehrdad, Frank W., Harsha, and many others.

I extend my heartfelt appreciation to my parents, Gyutaek and Haesook, as well as my sister, Inhee, for their continuous support and encouragement. Their support has been instrumental in this journey.

Last but not least, immense thanks go to my partner, Heebum, for standing by me, even in the most challenging times. His support was crucial, and I could not have completed this thesis without him.

Prior Publication

Parts of this thesis were previously published in conference papers [80, 99].

Contents

1 Introduction **11**

2 Breakwater: Overload Control for μs-scale RPCs **15**

2.1 Introduction 15
2.2 Motivation and Background 21
2.2.1 Problem Definition and Objectives 21
2.2.2 Overload Control in Practice 23
2.2.3 Locking Complicates Overload Control 27
2.2.4 Existing Overload Controls Cannot Handle Lock Contention 29
2.2.5 Challenges 32
2.2.6 Breakwater Approach 34
2.3 System Design 36
2.3.1 Overload Detection 37
2.3.2 Overload Control 43
2.3.3 Breakwater Client 50
2.4 Implementation 51
2.5 Evaluation 55
2.5.1 Evaluation Setup 56
2.5.2 CPU-bottlenecked Scenarios 59
2.5.3 Lock-bottlenecked Scenarios 77
2.5.4 Limitations of Breakwater 90
2.6 Discussion 93
2.7 Related Work 93
2.8 Conclusion 97

3 LDB: An Efficient Latency Profiling Tool for Multithreaded Applications 99
3.1 Introduction 99
3.2 Motivation 103
3.2.1 Debugging the Tail Latency 103
3.2.2 Challenges 107
3.3 System Design 108
3.3.1 Overview 108
3.3.2 Stack Sampling 111
3.3.3 Tracing Cross-thread Request Handling 115
3.3.4 Analysis Script 116
3.4 Implementation 117
3.4.1 LLVM-LDB 118
3.4.2 The LDB API and Parameters 119
3.5 LDB Use Cases 120
3.5.1 Reconstructing the Timeline of the Request 123
3.5.2 Tail Latency Debugging 124
3.5.3 Debugging Throughput of Qperf 129
3.6 Performance Evaluation 131
3.6.1 Microbenchmark 133
3.6.2 Portability of LDB 135
3.6.3 Overheads of LDB 136
3.6.4 Breakdown of LDB's overhead 139
3.7 Related Work 139
3.8 Conclusion 143

4 Future Work 145
4.1 Overload Control 145
4.1.1 Overload control for multi-layer services 145
4.1.2 Delay from other types of congestion 146
4.2 Tail Latency Profiling Tool 146
4.2.1 Distributed latency profiling 146
4.2.2 Detailed analysis of a function with high tail latency 146

5 Conclusions **149**

1 Introduction

In the last decades, a dramatic surge has been witnessed in the demand for applications that adhere to strict Service Level Objectives (SLOs), highlighting the crucial necessity for enhanced user interactivity and swift response times. For instance, artificial intelligence-enabled chatbots [98, 100] and search engines necessitate sub-second response times to ensure seamless user interactions. In the fast-paced world of real-time ad bidding [53, 90], the ability to determine participation and bid prices in mere milliseconds can make or break the deal. Likewise, high-frequency trading algorithms [25] are required to make split-second decisions and execute orders to capitalize on fleeting latency arbitrage opportunities.

Minimizing end-to-end latency is crucial for businesses, as it directly influences their revenue. Google, for example, revealed that a delay as small as 500 milliseconds could result in a significant 20% decline in their earnings [59]. Similarly, Amazon has indicated that for every 100 milliseconds of latency they encounter, their sales suffer a 1% reduction [73]. Additionally, an Akamai study demonstrated that for every 100 milliseconds increase in online retailer website load time, there could be a detrimental 6% decrease in sales [57]. Furthermore, the races for latency arbitrage occur approximately once every minute per symbol, with the winner outpacing the closest competitor by a mere 5-10 microseconds [77].

For better scalability, availability, modularity, and parallel processing,

implementation of datacenter applications often involves multiple RPCs. Each RPC now typically executes in mere microseconds, facilitated by in-memory operations, specialized hardware, and faster disks, leading to a high fan-out traffic pattern. With "high fan-out", tail latency takes on critical importance in attaining low end-to-end latency due to the *straggler problem*, where the completion of the response depends on the slowest RPC. In a complex RPC call graph utilizing microservice architectures, the impact of tail latency on the end-to-end latency becomes even more pronounced.

Considerable research and engineering effort has enhanced datacenter network performance through new hardware [38, 101], new flow control [104], new congestion control [17, 52, 54, 66], zero-copy within I/O [4, 44, 95], kernel bypass networking stack [1, 23, 34, 36], and hardware virtualization [31]. Thanks to these advancements, there has been a tenfold improvement in network bandwidth from 40 Gbps to 400 Gbps, while single-hop network latency has dwindled from tens of microseconds down to single-digit microseconds in the past ten years. In the meantime, CPU clock frequency has improved only 25% from 4 GHz to 5 GHz with the end of Dennard's scaling. As a result, congestion is now moving from networks to compute resources.

While there exist mechanisms to keep the latency low even with congested compute resources through hedging requests [27], employing Active Queue Management (AQM) [24, 42], and implementing admission control [8], these traditional solutions are primarily designed for millisecond-level RPC latency. Because of microsecond-level overhead per request, they are not entirely suitable for the current landscape of microsecond-scale RPCs.

This thesis explores three primary causes of tail latency in compute resources handling microsecond-scale RPCs and suggests systems to mitigate or diagnose them. First, we pinpoint that *receive livelock* significantly

bloats latency when CPUs are congested. When the server receives more load than its capacity while CPUs are congested, more CPU cycles are dedicated to packet processing and request parsing, leaving fewer for useful application logic. While the existing AQM mechanisms alleviate receive livelock for millisecond-scale RPCs by dropping them, they are insufficient for microsecond-scale RPCs due to the almost equivalent cost of dropping the request and request processing time. We identify that as request processing time reduces from milliseconds to microseconds, receive livelock re-emerges as a major latency source.

Second, we discovered that lock contention within an application, particularly blocking synchronization such as mutexes, contributes to tail latency because threads are delayed as they wait to acquire a contended lock. This lock contention causes threads to linger in the lock waiter queue, increasing tail latency. Determining whether the system should accept additional incoming load becomes complex in the face of lock contention. CPU-based signals like thread queueing delay or packet queueing delay prove ineffective, as threads waiting for locks do not consume any CPU cycles. Relying on end-to-end latency signals can be overly conservative as latency rises even when a single lock within a datapath becomes contended. While measuing per-lock waiter queueing delay does capture lock-specific congestion, it's unclear how to aggregate these per-lock delay signals to gauge the overall system's capacity to handle more incoming load.

Lastly, we attribute high tail latency to the application logic itself, such as performance bugs, inefficient algorithms, or a misalignment between the system design and the actual workload. Identifying the specific functions or modules responsible for high tail latency is challenging because most existing profiling tools focus on measuring CPU usage rather than latency, or they impose high overhead through static timestamp instrumentation.

To address CPU congestion and lock contention, we introduce Breakwater, a novel overload control for microsecond-scale RPCs featuring credit-based admission control and delay-based AQM. Breakwater decides whether to accept additional incoming load using one of two optional congestion signals: request queueing delay, which is rapid but only effective for CPU congestion, and performance-based efficiency metric, which operates more slowly but is applicable to both CPU congestion and lock contention. Our results demonstrate that Breakwater adeptly manages intermittent excess load, ensuring that tail latency remains within controlled bounds under both CPU congestion and lock contention.

To identify the specific functions responsible for high tail latency, we developed LDB, a tail latency debugging tool. LDB facilitates low overhead latency profiling without code modifications, based on the observation that the call stack remains the same while a thread is executing the same function. Additionally, by utilizing optional request tagging supplied by developers, LDB can reconstruct a detailed timeline for a request. This timeline includes intricate details such as context switches and inter-thread interactions, allowing for a more precise understanding of the latency behavior.

In the subsequent chapters, we will discuss Breakwater (Chapter 2) and LDB (Chapter 3) in detail. Chapter 4 discuss the future work, and Chapter 5 concludes with a summary of the key insights presented throughout the thesis.

2 Breakwater: Overload Control for μs-scale RPCs

2.1 Introduction

Modern datacenter applications are comprised of an interconnected set of microservices [26, 37, 103], utilizing Remote Procedure Calls (RPCs) to facilitate communication between them. These microservices often operate under stringent Service Level Objectives (SLOs), with some requirements even scaled down to microseconds with in-memory operations [29, 35, 43, 110], specialized hardware, and faster I/O. While recent advances in operating systems [76] and network hardware [107] have allowed these demanding SLOs to be met under typical load conditions, the challenge to adhere to these requirements becomes markedly difficult when the server's load nears or exceeds its capacity.

One of the key objectives of datacenter operators is to maximize the utilization of limited resources. Running a server near its full capacity can lead to maximum throughput, but it also increases the risk of high tail latency, particularly during transient overload situations when the load momentarily surpasses the server's capacity. Such overloads can be triggered by unpredictable changes in request arrival patterns, load imbalances, sudden

packet bursts, or redirected traffic due to a failure. Consequently, striking a balance between maintaining stringent SLOs and maximizing resource utilization becomes a new challenge. The challenge is further complicated by microsecond-level service times, as even slight delays or errors in making a decision can lead to long response times or suboptimal server utilization.

The primary objective of overload control is to shed excess load to ensure both high server utilization and low tail latency. Existing overload control schemes can generally be grouped into two main categories. One class of approaches involves dropping requests at the overloaded server or at its proxy [24, 42, 48], with the most optimistic scheduling leaving overload control task solely to the server.

Other schemes throttle clients' request sending rates [50, 61, 75] by requiring clients to probe the server's load. Unfortunately, neither approach is effective for handling microsecond-scale RPCs. When dealing with such very short requests, relying on the request drop is not practical, as the overhead is often on par with the service time of the request itself. On the other hand, client-side rate limiting requires a real-time understanding of server congestion to adjust rate limits accurately. However, synchronizing the server's congestion status between the server and the client requires a network round-trip time (RTT), and any delay in responding to congestion can be a substantial negative impact on performance of microsecond-scale RPCs.

Scaling the overload control system to accommodate a large number of clients adds another layer of complexity. The small resource requirements of a short RPC allow a single server to process millions of requests per second, potentially from thousands of clients [18, 62, 85]. In such a large-scale system, many clients have sporadic and infrequent demands on a particular server, making it challenging to set an accurate rate limit as the clients may

not have a fresh view of the server's congestion. Thus, server overload can be caused by "RPC incast" [8, 28], where a large number of clients make requests simultaneously, leading to a large queue build-up at the server. Some strategies involve probing the server directly before sending a request, but this can create significant overhead for microsecond-scale RPCs, especially when communication is needed for each individual request.

When multiple requests requires the same lock for processing, they contend for the lock, not CPU, introducing an additional source of delay in RPC server. However, devising a control signal that can effectively respond to lock contention is a compounding challenge. Since existing schemes that measure load or thread queueing delays are sensitive to overall system load, they fall short in identifying bottlenecks at the level of individual locks. Most existing schemes hinge on CPU congestion with queueing delay signal [70, 80] or end-to-end response time [8], but these methods falter under lock contention, especially with blocking synchronization (e.g., mutexes) that makes a thread yield the CPU rather than spinning on it (§2.2.3). Such contention results in lengthy lock acquisition wait times, thereby increasing tail latency and squandering valuable CPU resources.

To better understand the challenge of managing lock contention, consider a key-value store, where the key-value pairs are grouped together based on the hashes of their keys. Access to a bucket (i.e., a group of items with the same hash) is protected by a bucket lock. This means that in a key-value store, the number of locks corresponds to the number of buckets. However, a `GET` request acquires only a single lock which synchronizes access to the bucket holding the data it's accessing. As a specific piece of data becomes popular, the lock protecting its bucket becomes highly contended, negatively impacting the latency of all requests attempting to access that bucket. It is important to note that such contention and high delay impact *some but not*

all of the requests the application handles. The remainder of the requests can be accessing different buckets incurring no contention, finishing with minimal latency.

To maintain good performance under lock contention, one must reduce the load on the contended lock, and thus the latency of requests attempting to acquire it. On the other hand, this should not be done in a way that affects the throughput of requests not facing contention. The classic tension between throughput and latency is exacerbated in this case due to the unpredictability of request behavior: the locks accessed by a request can only be known after the execution of the request starts. Thus, the delay faced by different requests, that look identical when admitted to the server, can be very different depending on whether they attempt to access a contended resource or not. This renders overload signals that consider the overall delay of requests ineffective. Furthermore, blocking locks can prevent the load from saturating the CPU, rendering CPU congestion signals ineffective as well.

In this chapter, we introduce Breakwater, an overload control system for μs-scale RPCs, with particular attention to CPU congestion and lock contention. Breakwater adopts a balanced approach, situated between optimistic and skeptical scheduling. This "optimistic-enough" scheduling strategy aims to provide enough load to saturate the server but without incurring high latency. Any accidental delays that might arise from incorrect scheduling decisions are managed through a request drop with AQM to ensure low tail latency at all times.

Breakwater employs a server-driven admission control mechanism, wherein clients are permitted to send requests only after receiving credits from the server. Breakwater offers two modes of overload control signals: request queueing delay and performance-driven efficiency. The request queueing

delay signal is measured by the time from a packet arrival to a request execution. This mode is fast-acting but limited to CPU congestion scenarios. On the other hand, performance-driven efficiency is measured by the changes in system throughput relative to the changes in incoming load. While this mode is slower to respond, it is more versatile, capable of detecting a broader range of overload scenarios, including locking-related bottlenecks. If an overload signal indicates that the server is not currently overloaded, Breakwater issues more credits to the clients.

Breakwater minimizes the overhead of coordination (i.e., the communication overhead for the server to know which clients need credits) using *demand speculation*. In particular, a Breakwater server only receives demand information from clients when such information can be piggybacked on requests. When all known demand is satisfied, the server distributes credits randomly to clients. This approach does not require coordination messages to determine demand in clients. However, demand speculation can lead to issuing credits to clients who do not need them at that moment. These unused credits lower server utilization. Thus, Breakwater issues extra credits to ensure high utilization. Such overcommitment introduces the potential for queue buildup at the server if many clients with credits send requests simultaneously (i.e., RPC incast).

With RPC incast, the tail latency can grow beyond its SLO. In addition, because the performance-driven efficiency metric is a system-wide metric not considering the individual lock's contention, a hot lock can have a lengthy waiter queue, which further increases tail latency. To ensure tail latency stays below SLO even in such cases, we employ request drop with AQM as a safety net. Breakwater drops the request at request queues or lock waiter queues if it is expected to violate its SLO by waiting for the queue. In this way, Breakwater can offer the right load to maximize a server's throughput,

even if some requests must be aborted before/during processing.

We implemented Breakwater as an RPC library on top of the TCP transport layer. Our extensive evaluation of various workloads demonstrates that Breakwater achieves higher goodput with lower tail latency under CPU congestion and/or lock contention compared to SEDA [8] and DAGOR [70], the best available overload control systems. For example, Breakwater achieves 6.6% more goodput and 1.9× lower 99%-ile latency with clients' demand of 2× capacity, compared to DAGOR with a synthetic workload in a CPU congested scenario, and it achieves up to 1.6× more goodput with 5.7× lower 99th percentile latency compared to SEDA with a Memcached SET-heavy workload in a lock contended scenario. In addition, Breakwater scales to a large number of clients without degrading its benefits. For example, when serving 10,000 clients with Memcached, Breakwater achieves 14.3% more goodput and 2.9× lower 99%-ile latency than DAGOR. Compared to SEDA for the same workload, Breakwater achieves 5% more goodput and 1.8× lower 99%-ile when the clients' demand is 2× capacity.

Breakwater has some limitations. The performance can be degraded if there is a mismatch between the types of resources congested and the overload signal. It handles CPU congestion the best with request queueing delay signal and lock contention with the performance-driven efficiency signal. Furthermore, it requires application-level code changes to employ delay-based AQM on the locks and to provide the logic for how to handle the aborted request during processing (releasing acquired locks, cleaning up the state, freeing the resources allocated, etc).

2.2 Motivation and Background

2.2.1 Problem Definition and Objectives

Overload control is key to ensuring that backend services remain operational even when processing demand exceeds available capacity. Overload was identified as the main cause of cascading failures in large services [48]. Transient overload can occur for a variety of reasons. For example, it may not be cost-effective to provision enough capacity for maximum load [70]. Services can also experience unexpected overload conditions (faulty slow nodes, thermal throttling, hashing hot spots, etc.) despite careful capacity planning.

Without proper overload control, a system may experience congestion collapse producing no useful work as the majority of requests fail to meet their SLOs. Even when the average of clients' demand is less than the capacity, short-timescale bursty request arrivals can degrade latency for short requests. Microsecond-timescale RPCs are much more prone to performance degradation due to short-lived congestion than RPCs with longer service times [68].

RPCs with microsecond-scale execution time are prevalent in modern datacenters. Such RPCs span a variety of operations on data residing in memory or fast storage like M.2 NVMe SSDs (e.g., key-value stores [35, 110] or in-memory databases [29, 43]). The move towards microservice architectures has only increased the prevalence of such RPCs [26, 37, 103]. Further, a single server must process μs-scale requests at very high rates, possibly from thousands of clients [18, 62, 85]. To cope with μs-scale RPCs, an ideal overload control mechanism should provide the following properties:

1. No loss in throughput. An RPC server should be processing requests at its

full capacity regardless of overload, avoiding congestion collapse. Further, the overhead of performing the overload control must be minimal.

2. Low latency. An ideal overload control scheme should ensure that any request that gets processed spends minimal time queued at the server. Low queuing latency ensures that processed RPCs meet their SLOs, and is particularly important for μs-scale RPCs which tend to have tight SLOs.

3. Scaling to a large number of clients. For such short RPCs, clients with sporadic demand consume very little resources at the server. Thus, high server utilization requires scaling to a large number of clients. The ideal overload control system should be resilient to "incast" scenarios when a large number of clients send requests within a short period of time. In particular, overload control should prevent queue build-ups that result from incast without harming throughput.

4. Low drop rate. Dropping requests wastes resources at the server because it must spend time processing and parsing requests that will eventually be dropped. Furthermore, dropping requests harms the end-to-end tail latency of RPCs, especially when network round-trip time (RTT) is comparable to RPC execution time, making retries more expensive. Thus, overload control should minimize the drop rate at the server.

5. Fast feedback. Clients have more flexibility to decide the next action if they can discover when a request is unlikely to be served within its SLO. Thus, if a server expects a request will violate its SLO, it should notify the client as soon as possible so that it can decide an alternative action without having to wait for the request to timeout (e.g., giving up on the request, sending it to another replica, issuing a simpler alternative request, degrading the quality of the service, etc. [27]).

Next, we examine existing overload control mechanisms, which were de-

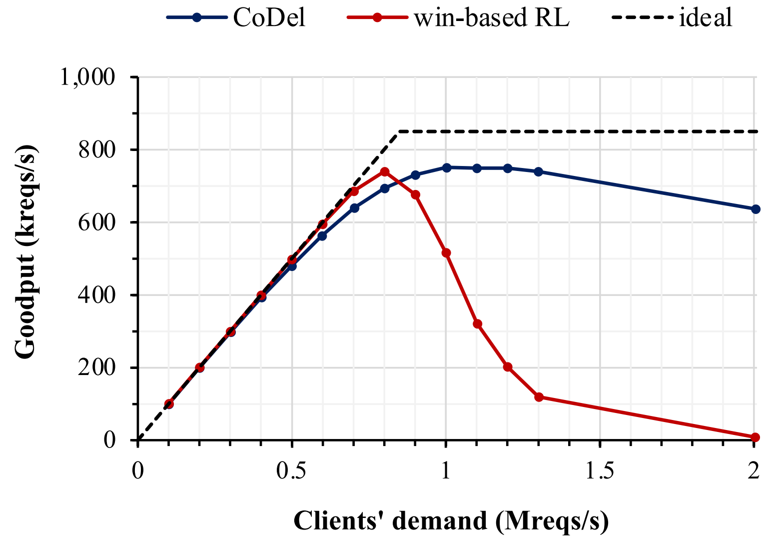
veloped for RPCs with relatively long execution times. Our goal is to understand the challenges of designing an overload control system for μs-scale RPCs.

2.2.2 Overload Control in Practice

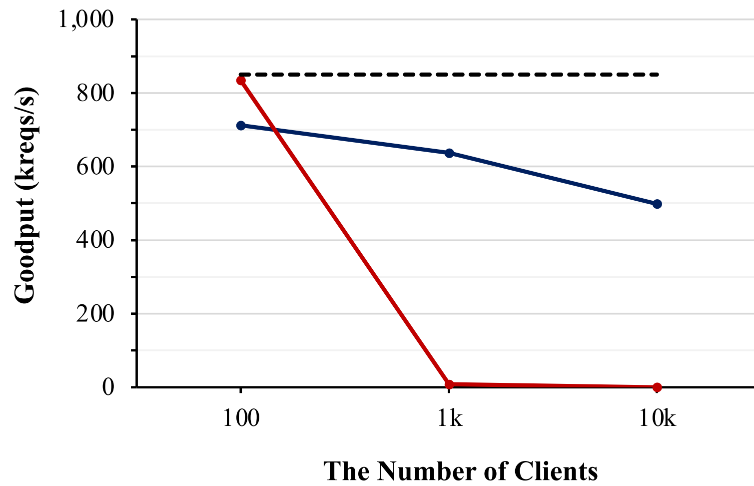
The fundamental concept in overload control is to shed excess load before it consumes any resources [2]. This is typically achieved by either dropping excess load at the server or throttling the sending rate of requests at the client. We look at the performance impairments of these two popular overload control approaches, developed for RPCs with long execution times, when used for μs-scale RPCs.

Active Queue Management (AQM). Such approaches operate as circuit breakers, dropping requests at a server or at a separate proxy under certain conditions of congestion. The simplest approach maintains a specific number of outstanding requests in the queue at the server, typically manually tuned by the server operator [42, 48, 86]. More advanced algorithms can improve performance and avoid the need for manual tuning. For example, CoDel maintains the queuing delay within a specific target value, dropping requests if the queuing delay exceeds the target [24, 42, 48]. RPC servers are typically required to report on success and on failure to avoid expensive timeouts [70, 86, 110]. This means that packets are processed, and failure messages are generated for dropped requests. This overhead is trivial when the message rate is low with a long execution time. However, it becomes a significant overhead in the case of μs-scale RPCs.

To demonstrate the limitations of the AQM approach, we implemented an RPC server that uses CoDel for AQM. Our main evaluation metric is the *goodput* of the server, defined as the throughput of requests whose response



(a) Goodput vs. clients' demand with 1,000 clients



(b) Goodput vs. the number of clients with clients' demand of 2M reqs/s

Figure 2.1: Goodput of CoDel and window-based rate limiting (win-based RL) with different clients' demands and different numbers of clients. It uses a synthetic workload of requests with exponentially distributed service time, with a mean of 10 μs.

time is less than the SLO. Figures 2.1 (a) and (b) demonstrate the goodput of CoDel with different clients' demands and different numbers of clients. This experiment uses a synthetic workload of requests busy-running for a exponentially-distributed service time, with a mean of 10 μs, making CPU congested as the clients' demand exceeds the server's capacity. The drop threshold parameter is tuned to achieve the highest goodput given an SLO of 200 μs. As the clients' demand increases and CPUs become congested, the system experiences receive livelock [2], where incoming requests are starved because the server is busy processing interrupts for new packet arrivals even though a majority of requests are dropped at the server. As a result, less CPU can be used for RPC execution, which leads to goodput degradation. Goodput degrades as the number of clients increases since there are fewer opportunities to coalesce failure messages, leading to larger overheads.

Client-side Rate limiting. In order to eliminate the overhead caused by dropping requests at the server, some overload control mechanisms limit the sending rate at the clients. With client-side rate limiting, clients probe the server, detect its capacity, and adjust their rate to avoid overloading the server [10, 50, 61, 75]. The reaction of clients to overload is delayed by a network RTT, which can lead to long delays when the execution time of RPCs is comparable to or less than the RTT. Further, the delay in getting feedback increases with the number of clients; consider the impact this has on overload control performance.

When the number of clients is small, the load generated by each individual client is large and each client exchanges messages with the server at a high frequency. This means that each client has a fresh view of the state of the server, allowing it to react quickly and accurately to overload. In this case, client-based approaches outperform AQM approaches because they have

fresh enough information to prevent overload at the server.

As the number of clients increases, the load generated by each client becomes more sporadic and messages are exchanged at a lower frequency between any individual client and the server. This means that in the presence of a large number of clients, each client will have a stale or inaccurate estimate of server overload, leading to clients undershooting or overshooting the available capacity at the server. When many clients overshoot server capacity, it can lead to incast congestion, causing large queueing delays. AQM avoids high tail latency by dropping excess load at the server, leading to AQM outperforming client-based approach for a large number of clients, despite having less than ideal goodput.

To illustrate the limitation of client-side rate limiting with μs-scale execution time, we implement window-based rate limiting used in ORCA [75]. The mechanism is similar to TCP congestion control. The client maintains a window size representing the maximum number of outstanding requests. Upon receiving a response, if the response time is less than the SLO, it additively increases the window size; otherwise, it multiplicatively decreases the window size. Figure 2.1 (a) and (b) depict the goodput of window-based rate limiting for exponentially-distributed service time of $10\,\mu$s (SLO $= 200\,\mu$s) on average. We optimized the parameters (i.e. additive factor and multiplicative factor) to achieve the highest goodput. Window-based schemes typically support a minimum of one open slot in the window (i.e., a minimum of one outstanding request at the server). This is problematic when there is a large number of clients as each client can always send one request, leading to incast and overwhelming the server. Rate-based rate limiting [10, 50] overcomes this limitation, but it still suffers from incast with a larger number of clients which results in high latency and low goodput.

Hybrid approaches that combine client-side rate limiting and AQM have

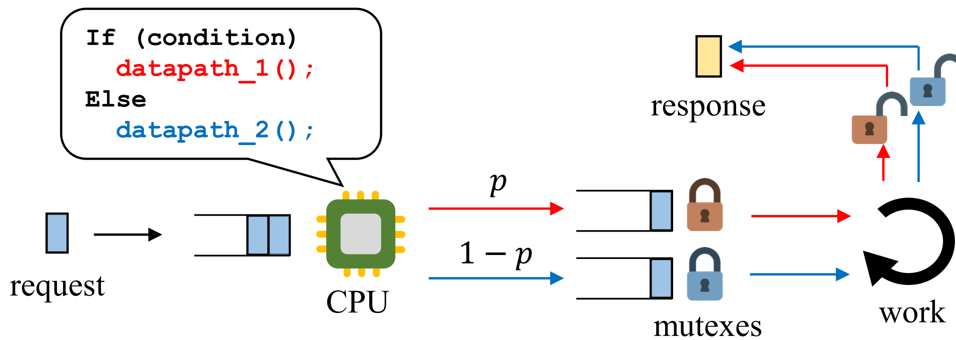


Figure 2.2: A simple example application with two global mutexes. With a probability p, the request takes the first data path (red arrow).

also been proposed. We provide a more comprehensive evaluation of rate-based rate limiting and hybrid approaches in §2.5.

2.2.3 Locking Complicates Overload Control

In modern datacenter applications, RPC requests often require blocking synchronization (e.g., mutexes, semaphores, and conditional variables) to serialize access to shared data. However, blocking synchronization primitives can experience contention when multiple requests attempt to access the same critical section, leading to a performance bottleneck. This is further complicated by the fact that the locks required by each request may be different depending on the request payload and the program's state. This makes it hard to know the data path a request will take before its actual execution.

The crux of this problem is that seemingly identical requests can have different execution paths at the server with different latency and throughput characteristics. This unpredictable behavior makes admission control hard,

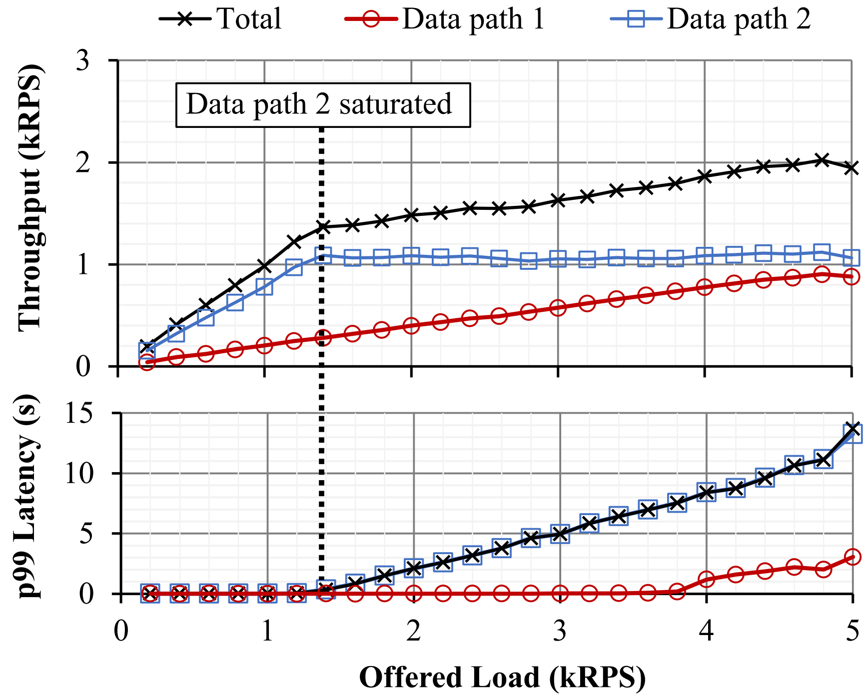


Figure 2.3: gRPC performance for the example application of Figure 2.2 (p = 20%). After acquiring a mutex, requests busy-loop for a time sampled from an exponential distribution with 1 ms average. Four cores are allocated for this experiment, one for each data path and two to adsorb any system overhead, ensuring that the CPU is not bottlenecked.

leading to the question: *which data path should admission control consider when admitting new requests?* To better understand this dilemma, consider the scenario in Figure 2.2. Incoming requests can take one of two paths, each protected with a different mutex. Requests can take the first data path with probability p, where $0 \leq p \leq 1$, and the second path with probability $1 - p$. We implemented this simple scenario in gRPC running on Linux. Figure 2.3 shows the performance of this scenario with $p = 20\%$ under various loads generated by client machines with an open-loop Poisson arrival process.

The existence of multiple data paths with different lock bottlenecks creates a dilemma. As shown in Figure 2.3, different datapaths are saturated at different offered load levels. Typically, clients and servers can't predict whether a request will take the datapath currently bottlenecked (data path 2 in the example). Here, the admission control dilemma emerges from the existence of multiple desirable operating points. If the operator desires low latency for all paths, then they have to sacrifice throughput, admitting only enough load to saturate the most congestion execution path (i.e., 1.2 kRPS in this example). On the other hand, if they desire high throughput, then they have to admit a high load and deal with the congested path through other means (e.g., dropping a request after admitting it). Next, we show that no existing overload control scheme can navigate this dilemma and produce good results in such scenarios.

2.2.4 Existing Overload Controls Cannot Handle Lock Contention

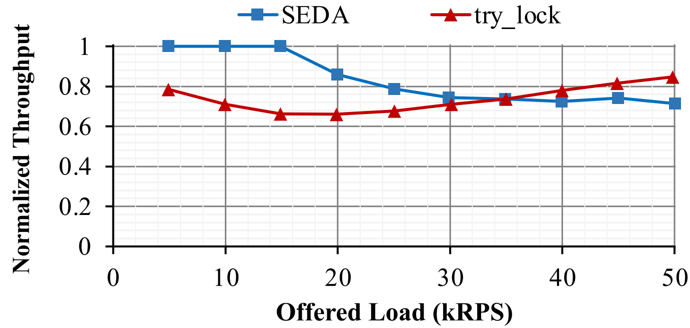
Overload control attempts to operate a server near its capacity with minimal SLO violations and request drops. The basic idea behind overload control is to keep track of the load on the server using a signal, adjusting the

admitted load based on that signal. Multiple signals have been proposed to improve the accuracy of admission control, including CPU utilization [61], end-to-end delay [8], and queuing delay [70, 80, 84]. However, none of these signals are useful in lock contention scenarios where the operator attempts to maximize throughput while maintaining low latency.

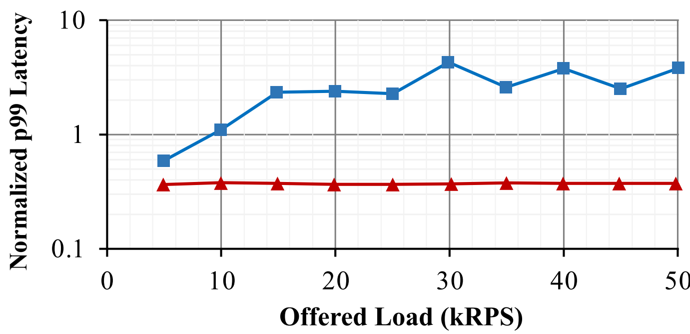
For example, Swift [84] and DAGOR [70] use past observations to predict the amount of queueing delay each request will face. However, in the presence of thousands of locks, it's unclear which queueing delay value (or statistic), if any, can be used to perform admission control. This is because admission control doesn't know in advance which locks requests will access, making it impossible to decide which value to react to without overestimating or underestimating overload. Note that any CPU-based metrics also fail as the CPU might not be the bottleneck in lock contention scenarios.

One possible approach to handle problematic or unpredictable lock behavior is to leverage existing primitives like `try_lock()` or `timed_mutex()`. Specifically, such primitives will allow requests to fail, avoiding latency, if the lock cannot be acquired due to congestion. However, overload control schemes that rely exclusively on request drops do not scale well due to the large overhead of packet drops. Furthermore, relying on existing primitives is not straightforward; `try_lock()` is a very aggressive overload control mechanism because it causes a request to fail on the first failed attempt to acquire a lock. On the other hand, `timed_mutex()` is too relaxed, forcing a request to wait for the full waiting time even under severe congestion conditions.

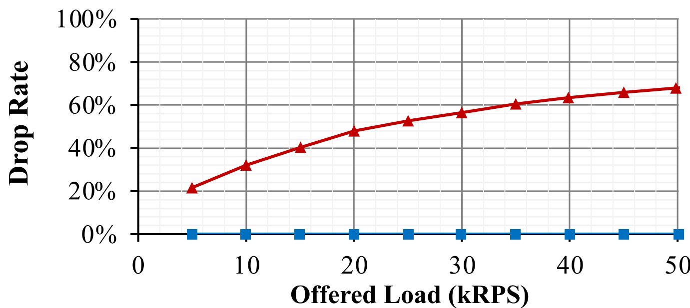
We demonstrate the limitation of existing overload control schemes, including the usage of `try_lock()`, by implementing those schemes for the scenario described in Figure 2.2, setting the average service time to 100 μs. However, rather than using gRPC, we use our implementation of SEDA on



(a) Normalized Throughput



(b) Normalized 99%-ile Latency



(c) Drop Rate

Figure 2.4: Performance SEDA and trylock for the example application of Figure 2.2 ($p = 20\%$) with 100 μs average service time on Shenango. Throughput and 99th percentile latency are normalized by the performance of Breakwater.

top of Shenango [76] for optimized network stack. It spawns a new thread per incoming request. We limit the number of spawned threads to bound the memory usage of the system. When a request is aborted, a failure message is reported to the client. The results are shown in Figure 2.4, comparing the throughput, tail latency, and drop rate of existing schemes, normalized by the performance of Breakwater.

As a client-side rate limiting approach, SEDA successfully bounds the tail latency as it throttles clients based on the measured tail end-to-end latency. However, by considering only the tail latency, it reacts to the most congested path, leading to poor throughput as it underutilizes the uncongested path. Sharing a similar principle as AQM, using `try_lock()` allows the system to achieve near-ideal latency while suffering from an extremely high drop rate and poor throughput. This is caused by `try_lock()`'s aggressiveness in dropping requests, wasting CPU and throughput even at low loads. Our proposal overcomes the shortcomings of existing systems, achieving the highest throughput while keeping the latency and drop rate low.

2.2.5 Challenges

Existing overload control schemes, developed for long RPCs, suffer significant performance degradation when handling μs-scale RPCs with CPU congestion and/or lock contention. The fundamental challenge facing existing schemes is the need for coordination of clients in order to schedule access to the server under very tight timing constraints with the appropriate congestion signal. This challenge is exacerbated by the following characteristics of modern datacenter applications:

1. *Short average service times.* We aim to support execution times for

RPCs on the order of microseconds. This requires devising an overload control scheme that can react at microsecond granularity while keeping coordination overheads significantly less than request service times. Achieving this compromise is challenging, and any errors in devising or implementing the overload control scheme can lead to either long queues and overload, or underutilization of the server.

2. Variability in datapaths and service times. RPC execution times typically follow a long-tailed distribution [27, 48, 71]. In addition, modern datacenter applications have thousands of data paths with different set of lock required. As the overload controller doesn't know which lock a request will require in advance, the lock queueing delay a request will face is unpredictable. The stochastic nature of RPC latency limits the accuracy of any coordination or scheduling at the client or server. Accurate scheduling requires knowledge of the execution time and the datapath of each request in advance, which is not possible because they often depend on the client's context, data in the request, and system's configuration. Further, this variability creates ambiguity for overload detection because a single request can be long enough to cause significant queueing delay.

3. No explicit signal to indicate server overload. With the lock contention, delay reflects the state of the most congested path, not the state of the overall system. On the other hand, CPU-related signal such as CPU utilization or thread queueing delay is not helpful when the bottleneck is not the CPU. Thus, we need a new approach to assessing the capacity of the server with different types of bottlenecks in order to make accurate admission control decisions.

4. Large numbers of clients. All previous challenges are exacerbated as the number of clients increases: accurate coordination becomes more challeng-

ing and overheads become higher (§2.5.2). Furthermore, a larger number of clients increases demand variability because it makes the system more susceptible to bursts (i.e., many clients generating demand simultaneously).

The challenges a server overload control system faces bear some similarities to those observed in network congestion control. At a surface level, network and compute congestion can be managed by similar mechanisms, but they each have fundamentally different requirements. Both are necessary to achieve good performance. Network congestion control aims to maintain short packet queues at switches while maximizing network link utilization. By contrast, overload control aims to maintain a short queueing delay at the RPC server while maximizing CPU utilization. There are two critical differences between these problems: (a) RPC processing often has high dispersion in request service times with diverse datapaths while packet processing times are almost constant, and (b) client-side demand can fluctuate more significantly at the RPC layer because clients may give up after a timeout or choose to send an RPC to a backup server. On the other hand, once a network flow starts, it generally completes. With such high variability in processing time, datapath, and demand, designing an overload control system requires overcoming different challenges than a network congestion control system.

2.2.6 Breakwater Approach

Breakwater begins with insights from receiver-driven mechanisms proposed in recent work on datacenter congestion control. In receiver-driven congestion control, a receiver issues explicit credits to senders for controlling their packet transmissions, which provides better performance than conventional sender-based schemes [54, 55, 66]. Inspired by this line of work, our design

has the following components:

1. Explicit server-based admission control: A client is only allowed to send a request if it receives explicit permission from the server. A server-based scheme allows for coordination that is based on the accurate estimation of the state of the server. Explicit admission control means that the load received by the server is completely controlled by the server itself. This allows for more accurate control that maintains high utilization and low latency. Server-based admission control can add an extra RTT for a client to request admission. We avoid this through piggybacking and overcommitting credits, as detailed later.

2. Demand speculation with overcommitment: The server requires knowledge of clients' demand in order to decide which client should be permitted to send requests. This is comparable to the need for clients to know about the state of the server in client-based schemes. Exchanging such information introduces significant overhead as the number of clients increases. Furthermore, as the execution time of RPCs decreases, the frequency of exchanging the demand information increases, further increasing overhead. The key difference between server-based schemes and client-based schemes is that we can relax the need for the server to have full information about clients' demand without harming performance. In particular, we allow the server to speculate about clients' demand and avoid lowering server utilization by allowing the server to overcommit, issuing more credits than its capacity.

3. AQM: Due to overcommitment, the server can occasionally receive more load than its capacity. In addition, with a lock contention, overload controller can admit more load even when the most congested datapath becomes congested to improve the server utilization. Thus, we rely on AQM to shed the excess load in both the request queues and the lock

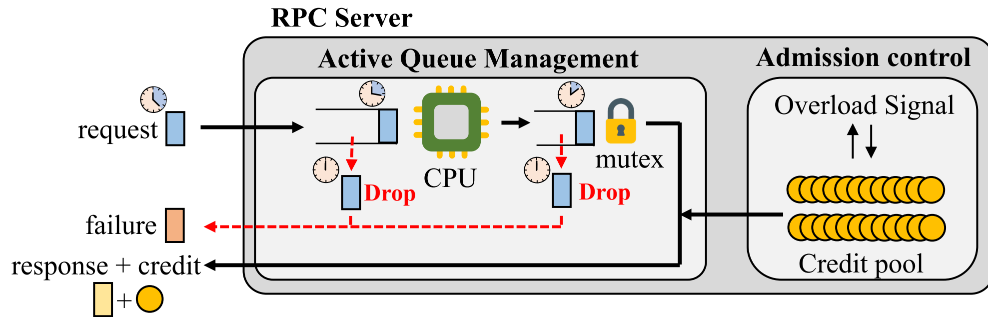


Figure 2.5: Breakwater Overview

waiter queues. In our scheme, a request is dropped at a last resort when it is expected to violate its SLO.

2.3 System Design

We present Breakwater, a scalable overload control system for μs-scale RPCs. Figure 2.5 depicts an overview of the operation of a Breakwater server. A new client joining the system sends a register message to the server, indicating the number of requests it has in its queue. The client piggybacks its first request to the registration message. The server adds the client to its client list, and if it is not overloaded it executes the request. The server then replies to the client with the execution result or a failure message. The server piggybacks with the response any credits it issued to the client depending on the demand indicated by the client. The client issues more requests depending on the number of credits it received. When the client has no further requests, it sends a deregister message to the server returning back any unused credits.

For the rest of the section, we present how Breakwater detects overload

and how it reacts to it. In particular, we present how a server determines the number of credits it can issue, how to distribute them among clients, and how clients react to credits or the lack thereof.

2.3.1 Overload Detection

Breakwater offers two distinct options for overload signals, each suited to different types of congestion. For CPU congestion, request queueing delay is utilized as an overload signal. It is both rapid and robust to noise, but it is confined solely to scenarios involving CPU congestion. As an alternative, performance-driven efficiency is employed to detect a broader spectrum of overload scenarios, including lock contention. While this is more versatile, accommodating more general overload conditions, it is comparatively slower and more susceptible to noise.

Detecting CPU Congestion with Reqeust Qeueuing Delay

There are multiple signals we can utilize to determine whether CPUs are congested. CPU load is a popular congestion signal—it is often used to make auto-scaling decisions in cloud computing [78]. However, using CPU utilization as a signal does not allow an overload controller to differentiate between the ideal scenario of 100% utilization with no delayed RPCs and a livelock state.

Another potential congestion signal is queue length at the server. A similar signal is widely used in network congestion control [17, 46]. Unfortunately, when RPC service times have high dispersion, queue length is a poor indicator of request latency. A more reliable signal is queuing delay, as it is accurate even under RPC service time variability. Furthermore, it is intuitive to map a target SLO to a target queueing delay at the server.

Thus, Breakwater uses queuing delay as its congestion signal to detect CPU congestion.

Effective overload control requires accurate measurement of the queuing delay signal. In particular, the signal should account for the sum of each of the queueing delay stages a request experiences, ignoring non-overload-induced delays. This ensures that the system only curbs incoming requests when it is overloaded. This is especially critical for microsecond-scale RPCs, as they leave little room for error.

Breakwater has two types of queues that grow with CPU congestion. Packets are queued while they await processing to create a request. Then, threads created to process requests are queued awaiting execution. Breakwater tracks and sums queuing delay at both of these. In particular, for every packet queue and thread queue, each item (e.g., a packet or a thread) is timestamped when it is enqueued. Each queue maintains the oldest timestamp of enqueued elements in a shared memory region, and this timestamp is updated on every dequeue. When the delay of a queue needs to be calculated, Breakwater computes it by taking the difference between the current time and the queue's oldest timestamp. We use this approach instead of measuring explicit delays of each request (i.e., the timestamp difference between request arrival and the request execution) to minimize noise in the queueing delay signal with interference, context switch, and interrupts.

There are multiple sources of delay that are not caused by CPU congestion. For example, long delays due to head-of-line blocking do not indicate a thread is waiting for resources, but rather it is a sign of poor load balancing. Accurate queueing delay measurement requires the system to avoid such delays. We find that the biggest source of such delays is the threading model used by the system. Our initial approach for developing Breakwater relied on the in-line threading model [32, 35] where a single thread han-

dles both packet processing and request processing. This choice was made as the in-line model provides the lowest CPU cost. However, it leads to head-of-line blocking as a single request with a large execution time can block other requests waiting at the same core. The alternative is relying on the dispatcher threading model [43] where a dispatcher thread processes packets and spawns a new thread for request processing incurring inter-thread communication overhead. However, this overhead is minimal when the dispatcher model is implemented using lightweight threads in recently proposed low-latency stacks (e.g., Shenango [76] and Arachne [67]). Thus, Breakwater employs the dispatcher model for request processing.

Detecting Lock Contention with Performance-driven Efficiency

There is a fundamental tradeoff between throughput and drop rate in the presence of unpredictable synchronization. To achieve high throughput, clients should offer enough load for the server to fully utilize its uncontended data paths. Unfortunately, this permits some congestion to occur in its contended data paths. Thus, our high-level strategy is to use an admission control scheme that admits enough load to keep all data paths operating at full capacity, combined with an Active Queue Management (AQM) mechanism that drops excess load on the contended data paths. This option of overload signal draws insight from network congestion control algorithms like PCC [39]. Specifically, it does not depend on a specific system's state. Rather, it observes the impact of its current admission rate on the behavior of the system, admitting more load only when it improves overall system performance.

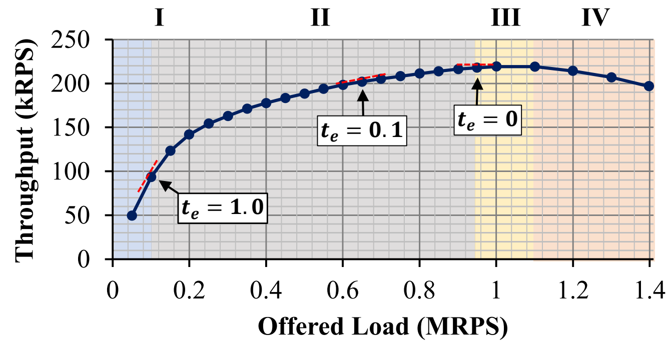
Our goal is to develop an admission control algorithm that allows a server operator to navigate the tradeoff between throughput and drop rate. Note

that the admission control algorithm should support scaling to a large number of data paths. Thus, we avoid developing an overload signal that has to take into account the state of every data path in the server.

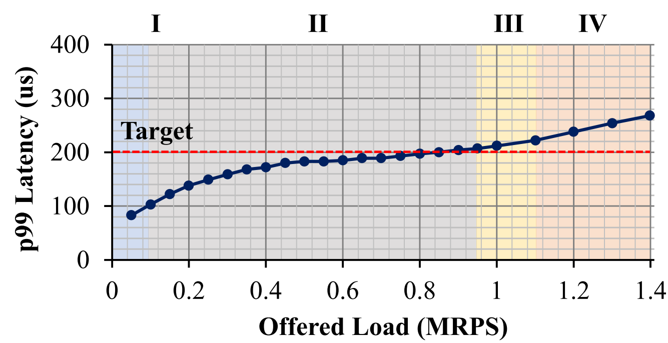
Intuition. To better understand the intuition behind our overload signal, we go back to the setup in Figure 2.2. Specifically, we rerun the experiment discussed in §2.2.4. However, we use a smaller service time per request (10 μs rather than 100 μs) because these results help to make our point clearer. Moreover, we don't use any admission control scheme but rely on the AQM scheme, discussed in the next section, to keep latency bounded. The results are shown in Figure 2.6. The design of our admission control scheme stems from observing that as the load increases, the system operates in four different phases:

Phase I (uncongested) is the phase where none of the locks or CPUs is congested. Throughput grows linearly with load increases because the system has capacity to handle all incoming demand. Further, tail latency increases only marginally because of bursts in the queue caused by the variable request arrivals, modeled as a Poisson arrival process. With no congestion, AQM does not drop the requests.

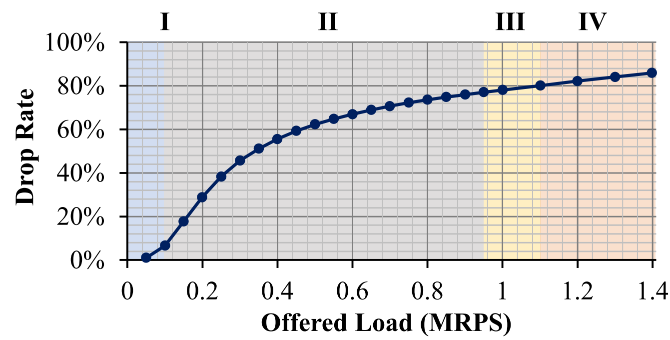
Phase II (partially congested) is the phase where a subset of locks are contended. As load increases, throughput increases sub-linearly because the system has capacity to handle only a fraction of incoming demand (i.e., the uncongested path still has capacity). Incoming requests that take the congested path will face high queueing delay, leading AQM to start dropping requests while keeping the tail latency near the target value. To generalize, different applications will produce a different concave line like that shown in Figure 2.6(a), where the slope of the curve decreases as more paths become congested. The exact shape of the curve depends on the number of congested paths, and their capacities along with the load.



(a) Throughput



(b) 99%-ile Latency



(c) Drop Rate

Figure 2.6: Performance of the application in Figure 2.2 (p = 20%) with 10 μs average service with the latency bounded by ASQM.

Phase III (congested) is the phase where all the data paths become congested. Thus, as the load increases, the throughput doesn't change. However, the increase in load increases CPU utilization because of the increase in network processing load and the increasing overhead of dropping requests. Eventually, the CPU also becomes congested, increasing tail latency.
Phase IV (congestion collapse) is the phase where the system enters a livelock state, spending more time dropping requests than processing them. During that phase, throughput degrades and latency keeps increasing.

Overview. Admission control should bound the incoming load to make the server operate in Phase II. Note that the values of latency, drop rate, and CPU utilization do not help identify the phase in which the server operates. However, by observing the slope of the throughput curve, one can identify the boundaries of Phase II. Specifically, Phase II starts when the slope of the throughput curve drops from 1 (i.e., the system can no longer handle *all* incoming requests) and ends when the slope reaches 0 (i.e., the system can no longer handle *any* additional incoming requests). A server operator that's interested in achieving a near-zero drop rate would operate the server at the leftmost edge of Phase II, where the slope of the throughput curve is slightly lower than one. On the other hand, a server operator that's interested in achieving the highest possible throughput would operate the server at the rightmost edge of Phase II, where the slope of the throughput curve is slightly higher than zero. The server operator can operate between those two points by choosing desired slope value. Additionally, the operator could specify the region of operation further by capping the maximum allowed drop rate.

We propose a performance-driven admission control algorithm with two parameters: efficiency threshold (t_e) and maximum drop rate (t_d). The efficiency threshold represents the target operating point on the through-

put curve in terms of the slope of the curve at that point. Specifically, t_e takes values between zero and one, with zero representing the highest possible throughput, and one representing zero drop rate. The maximum drop rate, t_d, allows a service operator to cap the drop rate at the expense of throughput to reduce the expected number of request drops. Breakwater uses the maximum drop rate in addition to the efficiency threshold to determine whether to accept more incoming load. Breakwater judges an RPC server to be overloaded, accepting no further load, if throughput improvement with additional load is less than the efficiency threshold or if the drop rate exceeds the maximum drop rate.

2.3.2 Overload Control

During overload, the system has to decide which requests to admit for processing and which requests to drop or possibly queue at the client. In this section, we explain our design for Breakwater's approach to overload control.

Server-driven Credit-based Admission Control

A Breakwater server controls the admission of incoming requests through a credit-based scheme. Server-driven admission control avoids the need for clients to probe the server to know what rate to send at. It also allows the server to receive the exact load it can handle. A credit represents availability at the server to process a single request by the client that receives the credit. A Breakwater server manages a global pool of credits (C_{total}) that is then distributed to individual clients. C_{total} represents the load the server can handle while maintaining its SLO.

With request queueing delay as the overload signal, Breakwater adjusts C_{total} such that the measured queuing delay (d_m) remains close to a target queuing delay (d_t) or measured efficiency value (e), which is set based on the SLO of the RPC. Every network RTT, Breakwater updates C_{total} based on the measured queuing delay (d_m). If d_m is less than d_t, Breakwater increases C_{total} additively.

$$C_{total} \leftarrow C_{total} + \mathcal{A} \tag{2.1}$$

Otherwise, it decreases C_{total} multiplicatively, proportional to the level of overload.

$$C_{total} \leftarrow C_{total} \cdot \max(1.0 - \beta \cdot \frac{d_m - d_t}{d_t}, 0.5) \tag{2.2}$$

Note that $\mathcal{A}$ controls the overcommitment and aggressiveness of the generation of credits. On the other hand, β controls the sensitivity of Breakwater to queue build-up. We explain how we select $\mathcal{A}$ and β in the next section.

With performance-driven efficiency as the overload signal, the server measures its efficiency (the change in throughput divided by the change in admitted load). If measured efficiency is less than the efficiency threshold (t_e), the server reduces the credit pool size, reducing the admitted load; otherwise, it increases the credit pool size. In particular, the server operates in iterations, each lasting a few end-to-end RTTs.[1] We measure the end-to-end RTT with the elapsed time between the credit issue and the successful response return which is tracked with an 8B unique credit ID. The server keeps track of the number of admitted requests from the current iteration and the previous iteration, in_{cur} and in_{last}, respectively. It also keeps track of the current throughput and the throughput in the previous iteration, out_{cur} and out_{last}, respectively. The efficiency metric

[1] We found that four RTTs allows for accurate measurement of all parameters while allowing for fast reaction to changes in the workload.

$e = (out_{cur} - out_{last})/(in_{cur} - in_{last})$ is compared to the efficiency threshold t_e. The server continuously monitors the drop count $drop_{cur}$ and decreases the admitted load if $drop_{cur}$ exceeds $t_d \cdot in_{cur}$. Breakwater uses additive increase / multiplicative decrease (AIMD) for credit management due to its simplicity. The details of the algorithm are shown in Algorithm 1.

Once C_{total} is decided, credits are distributed to clients. When C_{total} increases, new credits are issued to clients by piggybacking the issued credits to response messages sent to the clients. Explicit *credit* messages are only generated when piggybacking is not possible (i.e., server has no messages bound for the client). When C_{total} decreases, the server does not issue additional credits to the clients, or if the clients have unused credits, the server sends negative credits to revoke the credits issued earlier. The server can tell how many unused credits each client has by keeping track of the number of credits issued and the number of requests received. In the following section, we explain how Breakwater decides which client should be issued credits.

Demand Speculation with Overcommitment

There is a tradeoff between accurate credit generation and messaging overhead. Choosing which client should receive a credit can be simply determined based on the demand at the client. This requires clients to inform the server whenever their number of pending requests changes. The server can then select which clients to send a credit to based on demand. This ensures that all issued credits are used, allowing the server to generate credits that accurately represent its capacity. However, as we scale the number of clients, the overhead of exchanging demand messages overwhelms the capacity of the server.

Algorithm 1 Credit management with performance-driven efficiency

```
1: t_e: efficiency threshold
2: t_d: maximum drop threshold
3: C: the size of credit pool
4: in_{last,cur}: # of incoming requests in {last, current} iteration
5: out_{last,cur}: # of outgoing responses in {last, current} iteration
6: drop_cur: # of request drops in current iteration
7: a: increment step size
8: d: multiplicative decrement factor
9:
10: repeat Every 4 * end-to-end RTT
11:     if drop_cur > t_d · in_cur then
12:         C_total ← (1 − d) · C_total
13:     else if (in_cur − in_last)(out_cur − out_last) > 0 then
14:         if |out_cur − out_last| > t_e · |in_cur − in_last| then
15:             C_total ← C_total + a
16:         else
17:             C_total ← (1 − d) · C_total
18:         end if
19:     else
20:         C_total ← (1 − d) · C_total
21:     end if
22:     C_total ← max(C_total, C_min)
23:     C_total ← min(C_total, C_max)
24:     in_last ← in_cur
25:     out_last ← out_cur
26: until Application exits
```

In our design of Breakwater, we choose to eliminate the messaging overhead completely. A client notifies the server of its demand only if the demand information can be piggybacked on a request (i.e., the client already has a credit and can send a request to the server). The server therefore does not have accurate information about clients with sporadic demand as they can't update the server as soon as their demand changes. Thus, Breakwater speculatively issues credits based on the latest demand information even though it may be stale. Speculative generation of credits means that some clients that receive credits will not be able to use them immediately. If credits are generated to exactly match capacity, the server may experience underutilization because some credits are left unused when they are issued to clients with no queued requests. To achieve high utilization, speculative demand estimation is coupled with credit overcommitment to ensure that enough clients receive credits to keep the server utilized.

Overcommitment is achieved by setting the $\mathcal{A}$ and β parameters of the admission control algorithm. In particular, we set $\mathcal{A}$ to be proportional to the number of clients (n_c).

$$\mathcal{A} = \max(\alpha \cdot n_c, 1) \tag{2.3}$$

where α controls the aggressiveness of the algorithm. Further, each client is allowed to have more credits than its latest demand. The number of overcommitted credits per client (C_{oc}) is based on the number of clients (n_c), the total number of credits in the credit pool (C_{total}), and the total number of credits presently issued to clients (C_{issued}).

$$C_{oc} = \max(\frac{C_{total} - C_{issued}}{n_c}, 1) \tag{2.4}$$

The server makes sure that each client does not have unused credits more than its (latest) demand plus C_{oc} by revoking already issued credits if necessary.

Further, Breakwater attempts to avoid generating explicit credit messages whenever possible. This means that a new credit will be given to a client to whom the server is about to send a response unless that client has reached the maximum number of credits it can receive. Explicit credit messages are only generated when piggybacking a credit on a response is not possible. In the current version of Breakwater, the client that receives an explicit credit message is selected randomly, but we expect the selection could be smarter with per-client statistics. For example, the server can choose a client based on its average request rate to increase the likelihood of the client using the credit immediately.

AQM on Request Queues and Lock Waiter Queues: Active Synchronization Queue Management (ASQM)

The drawback of credit overcommitment is that the server may occasionally receive a higher load than its capacity, leading to long request queues. In addition, with admission control with performance-driven efficiency, the locks in the congested datapaths have long lock waiter queues. Therefore, we need a mechanism to drop the request in the queues as a last resort to avoid long queueing delays leading to high tail latency.

Breakwater assumes a standard queue abstraction per blocking synchronization object. However, to ensure scalability, Breakwater requires no coordination between queues, no per-queue parameter setting, and only minimal changes to the existing implementation of the synchronization API. Specifically, ASQM caps the total time a request is allowed to spend in a

queue, assigning each request a queueuing delay budget. The value of the budget represents the maximum queueing delay a request can tolerate for the server to respond within a target latency. The queueing delay budget is computed by subtracting the 99th percentile network latency and 99th percentile service time from the *target delay* of the request, leaving the slack time that the request can afford to spend in the server.

When a request arrives at the server, Breakwater assigns it a queueing delay budget. Before placing the request in each queue for a contended resource, it first checks the instantaneous queueing delay of the queue and drops the request if the queueing delay is larger than the request's remaining queueing delay budget. After the request is dequeued, it deducts the queueing delay it incurred from its budget. The queueing delay is measured by computing the difference between the current timestamp and the enqueue timestamp of the oldest item in the queue. In this chapter, we only consider the runnable thread queue in the CPU scheduler and the wait queues for blocking synchronization primitives. However, we believe the same idea can be applied to other queues for contended blocking interfaces such as blocking I/O.

Target delay vs. SLO. It's critical to note that the target delay used to compute the queueing delay budget is different from the RPC's Service Level Objective (SLO). The target delay is a per-server metric: a single server should finish a request or report failure within the target delay. On the other hand, an SLO is a per-request metric: a request of a specific type should finish within its SLO, taking into account that multiple attempts at multiple servers might be needed for the request to succeed. In Breakwater, the target delay is set by default to SLO divided by the maximum number of retries.

Handling dropped requests. Upon a request drop, the server returns

a failure message immediately to the client. At the server, a request drop incurs some CPU overhead to partially process the request and generate the failure message. Further, the failure message and retransmission of the request can incur networking overhead. If the overhead of dropping requests is large, a service operator can reduce the drop rate by choosing a higher value for the efficiency threshold (t_e), sacrificing throughput. At the clients, the dropped request may be handled in various ways: retransmission to another replica, triggering failure handling operations (e.g., online banking transaction), or degrading the quality of the response (e.g., search). For systems with replication and auto-scaling, retransmission is the most common failover mechanism. For the rest of this chapter, we focus on scenarios where an overloaded server has a non-overloaded replica which can serve dropped requests.

Retransmission of dropped requests introduces additional latency, inflating the overall delay faced by such requests, potentially harming their SLOs. Breakwater drops requests before they consume their delay budget. Thus, clients receive failure messages within the target delay. In the worst case, for each retransmission, a request will be delayed by at most the target delay (§2.5.3). Alternatively, if the SLO is tight, the client can send tied or hedged requests to multiple replicas to avoid the retransmission delay but incur the cost of coordination overhead and/or CPU wasted by duplicate executions [27].

2.3.3 Breakwater Client

Breakwater allows a client to queue requests if it does not have a credit for it. Client-side queuing is critical in a server-driven system as the client has to wait for the server to admit a request before it can send it. However,

if the client queue is too long, the request will experience high end-to-end latency. In Breakwater, in order to achieve high throughput and low end-to-end latency, we allow requests to expire at the client. The request expiration time is set based on its SLO.

When a client receives credits, it can immediately consume them if its queue length is equal to or larger than the number of credits it receives. Due to overcommitment, a client can receive credits which it cannot immediately consume (c_{unused}). When a client receives negative credits with decreased C_{total} at the server, the client decrements c_{unused}. However, if a client has already consumed all of its credits (i.e., $c_{unused} = 0$), no action is taken by the client.

2.4 Implementation

Breakwater requires a low-latency network stack in order to ensure accurate estimation of the queuing delay signal. This requires minimal variability in packet processing and no head-of-line-blocking between competing requests. We use Shenango [76], an operating system designed to provide low tail latency for μs-scale applications with fast core allocations, lightweight user-level threads, and an efficient network stack. Shenango achieves low latency by dedicating a busy-spinning core to reallocate cores between applications every 5 μs to achieve high utilization and minimize the latency of packets arriving into the server.

We implement Breakwater as an RPC library on top of the TCP transport layer. Breakwater handles TCP connection management, admission control with credits, and AQM on the request queue at the RPC layer. Furthermore, Breakwater extends Shenango's synchronization library to implement

ASQM, facilitating the adoption of Breakwater to Shenango applications. Breakwater abstracts connections and provides a simple individual RPC-oriented interface to applications, leaving applications to only specify request processing logic. Breakwater provides a single RPC layer per application (i.e., overload signal, credit pool, etc.) regardless of the number of cores allocated to the application and the number of clients of that application. A request arriving at a Shenango server is first queued in a packet queue. Then a Shenango kernel thread processes packets and moves the payload to the socket memory buffer of the connection. Once all the payload of a request is prepared in the memory buffer, a thread in Breakwater parses the payload to a request and creates a thread to process it. Threads are queued pending execution, and when they execute, they execute to completion.

Threading model. As explained earlier, Breakwater relies on a dispatcher threading model for accurate queueing delay measurement. A Breakwater server has a listener thread and the admission controller thread running. When a new connection arrives, the listener thread spawns a receiver thread and a sender thread per connection. Receiver threads read incoming packets and parse them to create requests. After parsing a request, AQM is performed, dropping requests if the current request queueing delay is greater than the request's latency budget. If a request is not dropped, the receiver thread spawns a new thread for the request. The new thread is enqueued to the thread queue. The sender thread is responsible for sending responses (either success or reject) back to the clients. If there are multiple responses, the sender thread coalesces them to reduce the messaging overhead. For all threads in Breakwater, we use lightweight threads provided by Shenango's runtime library.

Queueing delay measurement. Breakwater needs to measure instantaneous queueing delay to compare it against a request's remaining budget.

We instrument the request queues and the lock waiter queues to measure the queueing delay. When a thread is enqueued to a queue, Breakwater timestamps the request. When a queue is queried for the queueing delay, it returns the difference between the current timestamp and the enqueue timestamp of the oldest thread in the queue. Using an efficient hardware timestamp read function, Breakwater can measure the queueing delay with little overhead.

Performance-driven efficiency measurement. With the efficiency as the overload signal, Breakwater adjusts the credit pool size, once every iteration, based on five measures of efficiency and drop rate: in_{cur}, out_{cur}, $drop_{cur}$, in_{last} and out_{last}. The measures are updated (i.e., current measures are reset after their values are assigned to the last measures) after one end-to-end RTT from the time the credit pool size is updated to accurately reflect performance during an iteration. This period is selected because the incoming load changes in correspondence to the new pool size after at least one end-to-end RTT.

Lazy credit distribution. The admission controller updates C_{total} every RTT. Once the credit pool size is updated, the admission controller can re-distribute credits to clients to achieve max-min fairness based on the latest demand information. However, this requires the admission controller to scan the demand information of all clients, requiring $O(N)$ steps. To reduce the credit distribution overhead, Breakwater approximates max-min fair allocation with lazy credit distribution. In particular, Breakwater delays determining the number of credits a client can receive until it has a response to send to that client. The sender thread, responsible for sending responses to a client, decides whether to issue new credits, not to issue any credits, or to revoke credits based on C_{issued}, C_{total}, and the latest demand information. It first calculates the total number of credits the server should grant to client

x (c_x^{new}). If C_{issued} is less than C_{total}, c_x^{new} becomes

$$c_x^{new} = \min(demand_x + C_{oc}, c_x + C_{avail}) \tag{2.5}$$

where $demand_x$ is the latest demand of client x, c_x is the number of unused credits already issued to client x and C_{avail} is the number of available credits the server can issue ($C_{avail} = C_{total} - C_{issued}$). If C_{issued} is greater than C_{total}, c_x^{new} becomes

$$c_x^{new} = \min(demand_x + C_{oc}, c_x - 1) \tag{2.6}$$

The sender thread then piggybacks the number of credits newly issued for client x ($c_x^{new} - c_x$) to the response. It also updates c_x to c_x^{new} and C_{issued} accordingly.

Latency-aware Active Synchronization Queue Management (ASQM) API. Breakwater provides the following latency-aware APIs to enable ASQM:

```
bool mutex_lock_unless_congested(mutex_t *);
bool condvar_wait_unless_congested(condvar_t *, mutex_t *);
```

These interfaces are similar to those of a `try_lock()`, but their behavior is different. If the queueing delay of a blocking critical section exceeds a request's queueing delay budget, it returns false without waiting. Otherwise, it returns true after successfully acquiring the lock. An application developer can leverage the existing synchronization API provided by Shenango, including `mutex_lock()` and `condvar_wait()` for parts of the program that cannot handle dropping. For example, a maintenance thread running in the background may need to acquire a lock no matter how long it has to wait.

Identifying contended locks. In order to get the full performance benefits of Breakwater, developers must identify all the contended locks to

replace with Breakwater's ASQM APIs. A developer needs to hypothesize which locks are likely to be contended based on the application-specific knowledge and run experiments to verify which locks introduce a large queueing delay with per-lock queueing delay measurements. This process requires iterating multiple times until all the contended locks are identified and their code is modified to use the Breakwater API. Alternatively, a developer can use high-resolution latency profilers [91] to identify contended locks.

Application modification. Enabling Breakwater requires replacing blocking synchronization primitives with the ones provided in the Breakwater API. Further, Breakwater allows requests to be dropped after they have been partially processed by the server, potentially modifying some states or reserving some resources. Thus, enabling Breakwater requires the application to perform all necessary clean-up after a request is dropped (e.g., freeing memory it allocated to the request and releasing other locks the request currently holds). However, the complexity of handling request drops can be significantly reduced by utilizing features of modern programming languages, such as RAII in C++ with smart pointers and scoped locks.

2.5 Evaluation

Our evaluation answers the following questions under two different bottlenecks: CPU congestion and lock contention. With CPU congestion scenarios, we use request queueing delay as the overload signal (§ 2.5.2); with lock contention scenarios, we use performance-driven efficiency (§ 2.5.3) unless otherwise noted.

1. Does Breakwater achieve the objectives of overload control defined in

§2.2 even given tight SLOs?

2. How much code change is required?
3. Can Breakwater maintain its advantages regardless of load characteristics (i.e., average RPC service time and service time distribution)?
4. Can Breakwater effectively handle CPU congestion and lock contention within an application?
5. Can Breakwater scale to large numbers of clients?
6. Can Breakwater react quickly to a sudden load shift?
7. What is the impact of Breakwater's key design decisions: demand speculation and credit overcommitment?
8. How sensitive is Breakwater's performance to different parameters?
9. What are the limitations of Breakwater?

2.5.1 Evaluation Setup

Testbed: We use 11 nodes from the Cloudlab xl170 cluster [72]. Each node has a ten-core (20 hyper-thread) Intel E5-2640v4 2.4 GHz CPU, 64 GB ECC RAM, and a Mellanox ConnectX-4 25 Gbps NIC. Nodes are connected through a Mellanox 2410 25 Gbps switch. The RTT between any two nodes is 10μs. We use one node as the server and ten nodes as clients. The server application uses up to 10 hyper-threads (5 physical cores) for processing requests, and the client application uses up to 16 hyper-threads (8 physical cores) to generate load. All nodes dedicate a hyper-thread pair for Shenango's IOKernel.

Workloads: We evaluate Breakwater under CPU congestion (§ 2.5.2) and lock contention (§ 2.5.3) using three applications: 1) a synthetic application with its execution time drawn from an exponential distribution, 2) Memcached, a latency-sensitive in-memory key-value store that exhibits

both locking bottlenecks and CPU bottlenecks, and 3) Lucene, a search application with significant lock contention overhead.

Baseline. We compare Breakwater to DAGOR [70] and SEDA [8]. DAGOR is a priority-based overload control system used for WeChat microservices. Priorities are assigned based on business requirements across applications and at random across clients. We only consider a single application in our evaluation. DAGOR uses queueing delay to adjust the priority threshold at which a server drops incoming requests (i.e., requests with a priority lower than the threshold are dropped). To reduce the overhead of dropped requests, the server advertises its current threshold to clients, piggybacked it in responses. Clients use that threshold to drop the requests. Note that DAGOR does not drop its threshold to zero, meaning that a request with the highest priority value (i.e., a priority of one) will never be dropped. SEDA uses a rate-based rate limiting algorithm. It sets rates based on the 90%-ile response time. Since we evaluate the performance of Breakwater using the 99%-ile latency metric, we modified SEDA's algorithm so that it adjusts rates based on 99%-ile response time. We implement DAGOR and SEDA as an RPC layer in Shenango with the same dispatcher model as Breakwater.

Setting end-to-end SLO. We set tight SLOs to support low-latency RPC applications. We budget SLOs based on the server-side request processing time and the network RTT. An SLO is set as $10\times$ the sum of the average RPC service time measured at the server and the network RTT; the multiplicative factor of 10 was inspired by recent work on μs-scale RPC work [60, 71]. The RTT in our setting is $10\,\mu$s, leading to SLOs of $110\,\mu$s, $200\,\mu$s, and 1.1 ms for workloads with $1\,\mu$s, $10\,\mu$s, and $100\,\mu$s average service times, respectively. These are comparable with SLO values used in

practice [41].

Evaluation metrics: We report goodput, 99%-ile latency, drop rate, and reject message delay. Goodput represents the number of requests processed per second that meet their SLO. Reported latency captures all delays faced by a request from the moment it is issued till its response is received by the client. This includes any queuing delay at the client, communication delay, and all delays at the server. We report the drop rate at the server only, as it is the factor that directly impacts overall system performance. *Note that SEDA does not support any AQM at the server and has zero drop rate in all experiments.* Reject message delay represents the delay between the departure of a request from a client and the arrival of a reject message back to the client when that request is dropped at the server.

Parameter tuning. We tune the parameters of all systems so that they achieve the highest possible goodput. We re-tune the parameters when we change the average service time, service time distribution, and the number of clients. Note that Breakwater and DAGOR do not require parameter re-tuning for a different number of clients while SEDA does. Specifically, we need to scale adj_i parameter in SEDA based on the number of clients to get the best goodput.

For Breakwater, we set $\alpha = 0.1\%$, $\beta = 2\%$. With the overload signal of request queueing delay, we use d_t to 40% of SLO (e.g., $d_t = 80\mu$s for exponential service time distribution with 10μs average and 200 μs SLO), With the overload signal of efficiency, we use an efficiency threshold (t_e) of 10%, a maximum drop rate (t_d) of 100%. We determine the queueing delay budget for ASQM by deducting 99th percentile service time and 99th percentile network delay (20 μs) from the target delay for each workload.

For DAGOR and SEDA, which are devised for ms-scale RPCs, we scale

down the hyperparameters from the default values. For DAGOR, we update the priority threshold every 1 ms (instead of 1 s) or every $2,000$ requests and use $\alpha = 5\%$ and $\beta = 1\%$. We assign random priority for each request ranging from 1 to 128, which is the default priority setting with one type of service in DAGOR [70]. We tune $DAGOR_q$ for each workload (e.g., $DAGOR_q = 70\mu$s for exponential service time distribution with 10μs on average). For SEDA, we used the same default parameter from [8] except for $timeout$, adj_i, and adj_d. We set $timeout = 1$ ms (instead of 1 s) and tune adj_i and adj_j for each workload (e.g., $adj_i = 40$, $adj_d = 1.04$ for exponential workload with 10μs average with 1,000 clients). AQM in Breakwater and DAGOR drops requests right after parsing packets to requests, following the drop-as-early-as-possible principle [2]. We run all the experiments for four seconds. We measure steady state performance with converged adaptive parameters by collecting data two seconds after an experiment starts.

2.5.2 CPU-bottlenecked Scenarios

In this subsection, we examine the scenarios where the CPU becomes the bottleneck of the application. To address CPU congestion promptly and effectively, we use request queueing delay as the congestion signal for admission control.

Performance for Synthetic Workload

Workload: We run 1,000 clients divided equally between the ten nodes in our CloudLab setup. We generate the workload with exponential, constant, and bimodal service time distributions with $1\,\mu$s, $10\,\mu$s, and $100\,\mu$s average where each client generates the load with an open-loop Poisson process. We change the demand by varying the average arrival rate of requests at

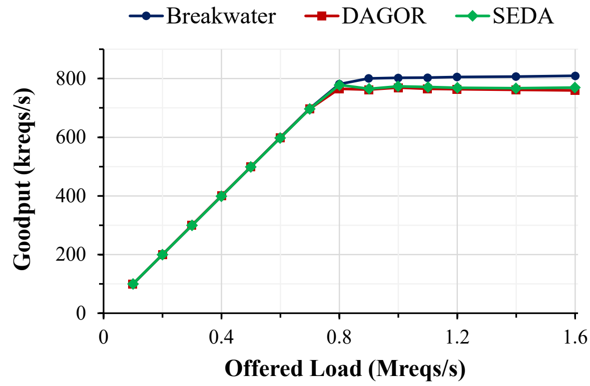
the server between 0.1× to 2× of server capacity. Exponential service time distribution models applications waiting for a shared resource while busy-spinning; constant distribution models applications with a fixed amount of latency such as fetching value from memory or flash drive; bimodal distribution models applications that cache frequently requested values, which will have shorter execution time compared to non-cached results. In particular, 20% of the requests take four times the average service time, and 80% of the requests take one fourth of the average following the Pareto principle.

Overall performance: Figure 2.7 shows the performance for a workload whose service time follows an exponential distribution with 10 μs average. The capacity of the server in this case is around 850k requests per second.

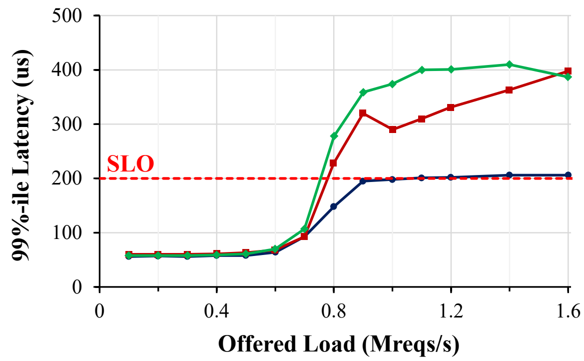
When the clients' demand is less than the capacity, all three systems perform comparably in terms of goodput, latency, and drop rate. The only noticeable difference among them is that, at 700k reqs/s, SEDA has a 15% higher 99%-ile latency than Breakwater or DAGOR. This is because SEDA doesn't drop requests at servers.

When the clients' demand is around the capacity of the server, Breakwater achieves 801k requests per second for goodput (or 808k reqs/s of throughput), which is around 5% overhead when compared to the maximum throughput with no overload control. Other systems have higher overhead than Breakwater.

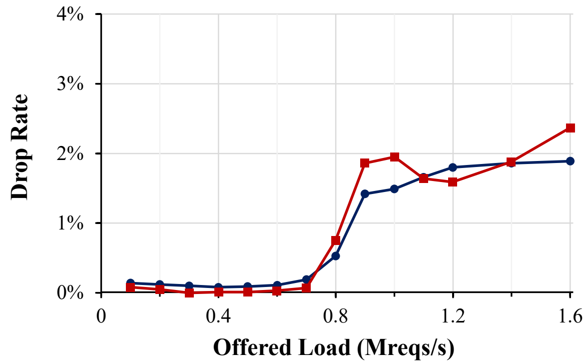
When the demand exceeds the capacity, incast becomes the dominant factor impacting performance. Breakwater handles incast well by preventing clients from sending requests unless they have credits, limiting the maximum queue size. Thus, Breakwater achieves higher goodput with lower and bounded tail latency. On the other hand, SEDA experiences high tail latency because clients do not coordinate their rate increase, making multiple clients increase their rate simultaneously and overwhelm the server. De-



(a) Goodput



(b) 99%-ile Latency



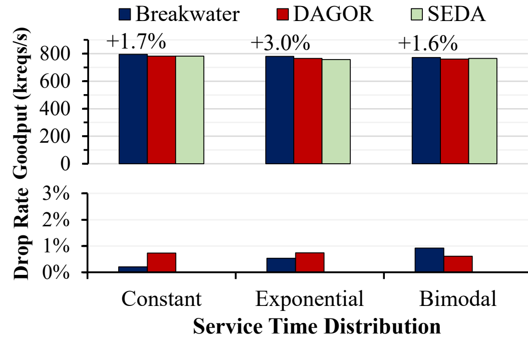
(c) Drop Rate

Figure 2.7: Performance of Breakwater, DAGOR, and SEDA for synthetic workloads with the exponential distribution of 10μs average.

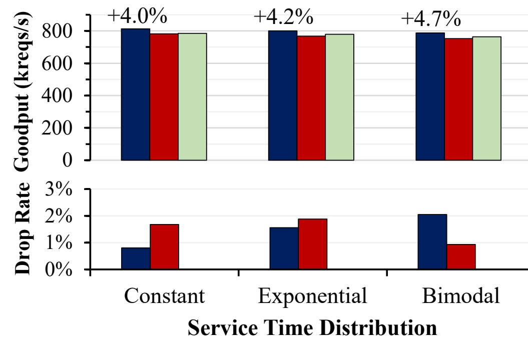
layed reaction to overload does not allow SEDA to react quickly to incast. DAGOR's high tail latency is also explained by delayed reaction as it updates its priority threshold every 1 ms or every 2,000 requests. Breakwater is also impacted by incast due to the overcommitted credits, which lead to increased tail latency and higher drop rate with overload. However, Breakwater relies on delay-based AQM which effectively bounds the tail latency while maintaining a comparable drop rate to DAGOR.

Impact of Workload Characteristics: To verify that Breakwater's performance benefits are not confined to a specific workload, we repeat the experiments with different service time distributions and different average service time values. Figure 2.8 shows goodput and drop rate with three different distributions of the service time whose average is 10μs, where the load generated by 1,000 clients is 0.9×capacity, 1.2× capacity, and 2× capacity. The service time distributions are aligned over the x-axis in ascending order of variance. Breakwater achieves the highest goodput regardless of the load and service time distribution. All three systems experience small goodput reduction with higher variance, especially when the load is 2× the server capacity. The goodput reduction of DAGOR and SEDA comes from their poor reaction to incast, whose size increases as the load increases. As a result, Breakwater's goodput benefit becomes larger as the clients' demand increases. Breakwater achieves 5.7% more goodput compared to SEDA and 6.2% more goodput compared to DAGOR with exponential distribution at a load of 2× capacity. With a higher variance of the service time distribution, the drop rate of the Breakwater tends to increase because a larger number of credits are overcommitted with higher variance, but it is still comparable to DAGOR.

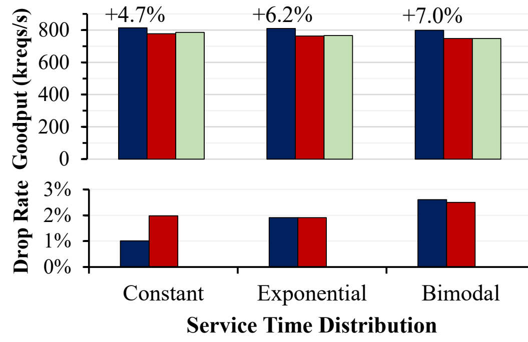
Figure 2.9 depicts performance with an exponential service time distribution and different average service times with 1,000 clients. Breakwater



(a) demand = 0.9× capacity

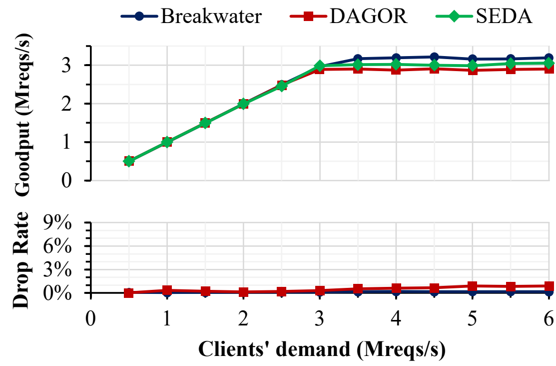


(b) demand = 1.2× capacity

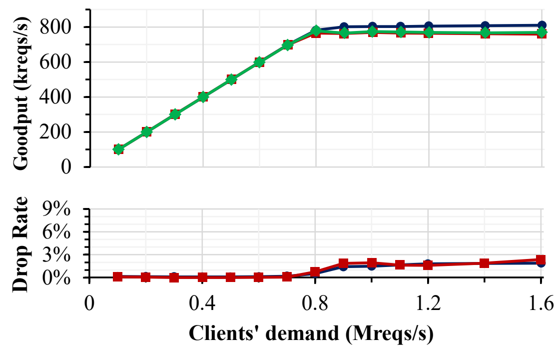


(c) demand = 2× capacity

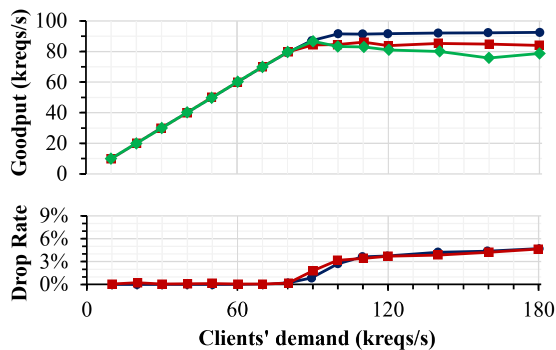
Figure 2.8: Goodput and drop rate with different service time distribution of 10μs average with 1,000 clients (The label represents the goodput gain compared to the worst of baselines).



(a) 1 μs



(b) 10 μs

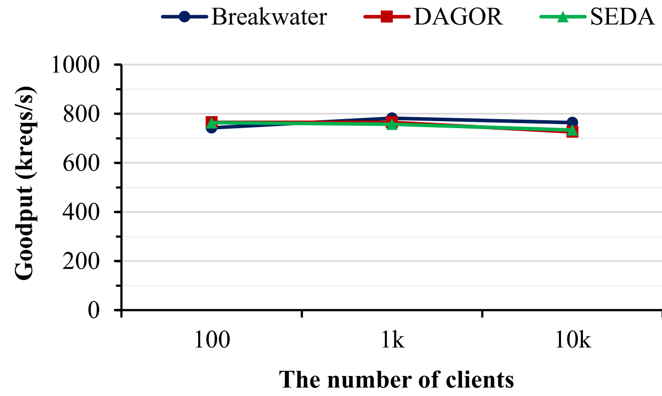


(c) 100 μs

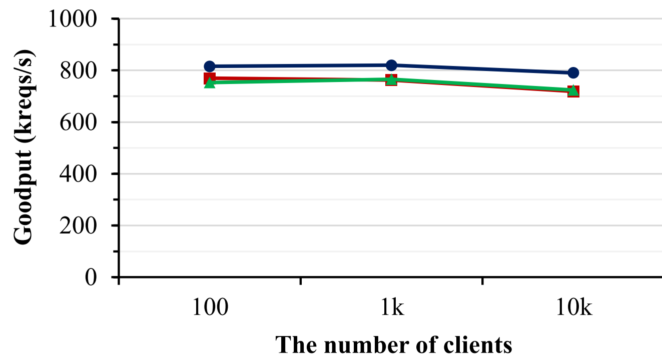
Figure 2.9: Goodput and drop rate with different average service time with 1,000 clients.

outperforms DAGOR and SEDA regardless of the clients' demand and the average service time. As the average service time increases, clients and servers exchange messages less frequently, exposing the delayed reaction problem in SEDA and DAGOR. With short service times (i.e., 1μs), clients and servers exchange messages very frequently, giving clients a fresh view of the state of the server in case of DAGOR and SEDA, allowing clients to react quickly to overload. With high demand, the size of incast gets larger which is poorly handled by SEDA and DAGOR. With clients' demand of 2$\times$ capacity with 100μs (i.e., 180k reqs/s), Breakwater achieves 17.5% more goodput than SEDA and 10.2% more goodput with a comparable drop rate compared to DAGOR.

Scalability to a Large Number of Clients: We vary the number of clients from 100 to 10,000 with synthetic workload whose service time follows exponential service time distribution of 10μs average. Note that the server capacity is around 850k requests per second. Figure 2.10 depicts the goodput with different numbers of clients. As clients' demand nears and exceeds the capacity, the goodput of all systems degrades as the number of clients increases. As the number of clients increases, the size of the incast increases, leading to performance degradation. This is problematic for Breakwater as well since overcommitment can occasionally result in large bursts of incoming requests. The performance of DAGOR and SEDA drops more than Breakwater as the number of clients increases. This is because each client exchanges messages with the server less frequently as the number of clients increases. The stale view of the server status leads clients to overwhelm the server. Note that for SEDA's best performance, we scale the additive rate increase factor (adj_i) to the number of clients. This helps mitigate any bursty behavior that can result from multiple clients sharply increasing their rate simultaneously. A small increase factor is not practical



(a) demand = capacity



(b) demand = 2× capacity

Figure 2.10: Goodput with different numbers of clients for exponential workload with 10 μs average service time

for a small number of clients as it will lead to slow ramp-up of rates after an overload, leading to lower utilization of the server. Because of this issue, SEDA has a much slower convergence time to the right rate, making it impractical for load shift scenarios as we show next.

Further, it is hard to tune SEDA dynamically. The rate control algorithm in SEDA is implemented at the client, and dynamic tuning requires each client to know the total number of *active* clients. Such a dynamic approach will lead to performance degradation as the client will retune its parameter to at least an RTT after the number of clients changes. The drawbacks of such a delayed reaction can be seen in the behavior of DAGOR. Further, exchanging such information might not feasible in practice due to messaging overhead as well as privacy concerns (e.g., a FaaS cloud provider will not want any of its clients to know the total number of clients). Note that even though Breakwater also scales the number of newly issued credits to the number of clients (Equation 2.1 and 2.3), Breakwater is server-driven, and the server has perfect knowledge of the number of active clients at all times with no need to expose this information outside. In SEDA, by contrast, each client cannot have perfect knowledge of the number of active clients. Each client would have to guess or receive feedback from the server to scale the increment factor.

Reaction to Sudden Shifts in Demand: An RPC server may experience sudden shifts in demand for many reasons, such as load imbalance, packet bursts, unexpected user traffic, or redirected traffic due to server failure. To verify Breakwater's ability to converge after a shift in demand, we measure its performance with a shifting load pattern. We use a workload whose service time follows an exponential distribution with 10μs average and calculate goodput, 99%-ile latency, and mean reject message delay every 20 ms. When the experiment starts, 1,000 clients generate requests at

400k reqs/s (0.5× capacity). Then, clients double their request rate to 800k reqs/s (0.9× capacity) at time = 2 s, then triple their demand to 1.2M reqs/s (1.4× capacity) at time = 4 s. Clients sustain their demand at 1.2M reqs/s for 2 seconds. Then, clients reduce their demand back to 800k reqs/s at time = 6 s and finally to 400k reqs/s at time = 8 s. Figure 2.11 depicts a time series behavior of all systems.

When the clients' demand is far less than the capacity, all three overload control schemes maintain comparable goodput and tail latency at a steady state. When demand increases to near server capacity, Breakwater converges fast, exhibiting a stable behavior in terms of both goodput and tail latency. On the other hand, DAGOR and SEDA experience higher tail latency because of the poor reaction to the transient server overload. As the server becomes persistently overloaded with a sudden spike at time = 4 s, Breakwater converges quickly while DAGOR and SEDA suffer from congestion collapse. Breakwater experiences a momentary tail latency increase (reaching 1.4× the SLO) with the sudden increase of clients' demand due to more incast caused by overcommitted credits. However, credit revocation and AQM rapidly limit the impact of any further incast. When demand returns back below the capacity at time = 6 s, Breakwater doesn't show a noticeable goodput drop while the DAGOR and SEDA experience a temporary goodput drop down to 77.5% and 82.6% of the converged goodput, respectively.

SEDA reacts slowly to the demand spike since each client needs to wait for a hundred responses or 1 ms to adjust its rate. After the demand spikes beyond the capacity, the server builds up long queues, and the latency goes up beyond SLO, resulting in almost zero goodput. SEDA takes around 1.6 s to recover its goodput. DAGOR also has the delayed reaction problem, but its goodput converges more quickly than SEDA thanks to AQM, taking

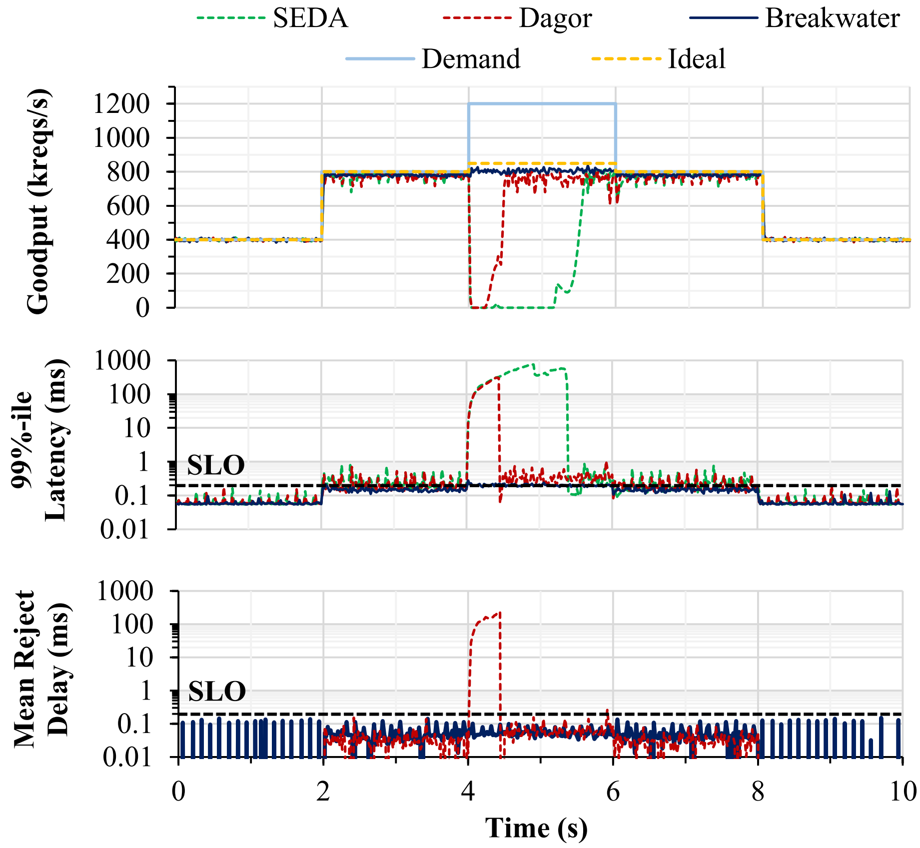
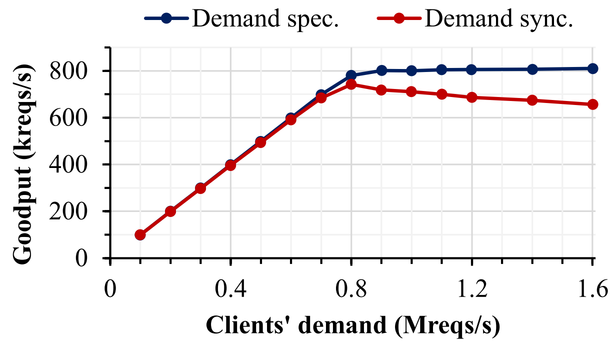


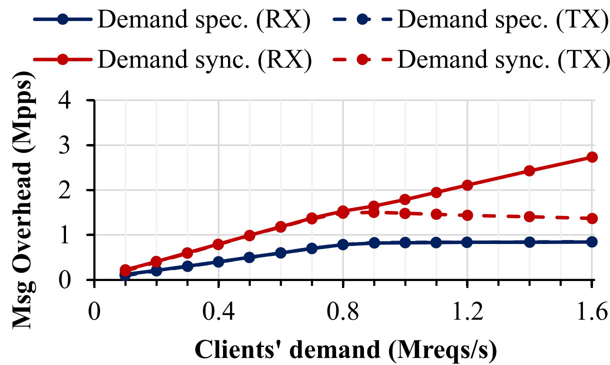
Figure 2.11: Goodput, 99%-ile latency, and mean rejection delay with a sudden shift in demand with 1,000 clients.

500 ms to recover its goodput. During the congestion collapse period, the 99%-ile latency of DAGOR soars up to 300 ms and its mean delay of reject message reaches 220 ms. This is problematic as clients cannot receive the feedback in a timely manner, making them rely on expensive timeout.

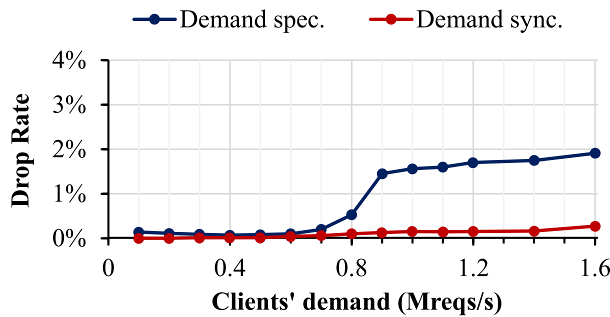
The Value of Demand Speculation: To quantify the performance benefits of demand speculation, we compare the two strategies for collecting demand information: demand synchronization and demand speculation. With demand synchronization, clients notify the server whenever their demand changes using explicit demand messages, and the server generates explicit credit messages to clients if it cannot be piggybacked to responses. With demand speculation, the server speculatively estimates client demands based on the latest demand information and piggybacks credits to the responses as much as possible. The load is generated by 1,000 clients where the service time per request follows an exponential distribution with an average of 10μs. The message overhead is measured by the number of packets received (RX) and sent (TX) at the server. With demand synchronization, both RX and TX message overhead increase as the clients' demand increases, leading to goodput degradation (Figure 2.12 (a)). In particular, as shown in Figure 2.12 (b), explicit demand and credit messages double RX and TX message overhead below and at the capacity (i.e., 850k requests per second). As the system gets overloaded, the overhead of demand messages keeps increasing because per-client demand changes more frequently with increased clients' demand. Further, the overhead of generating credits contributes to the cost of synchronization. The server sends more credit messages during low demand as they cannot be piggybacked on responses due to low request rates. As the load increases beyond capacity, more credits can be piggybacked to the responses, which results in the reduction of TX overhead. Demand synchronization has a smaller number of overcommitted credits,



(a) Goodput



(b) Message Overhead



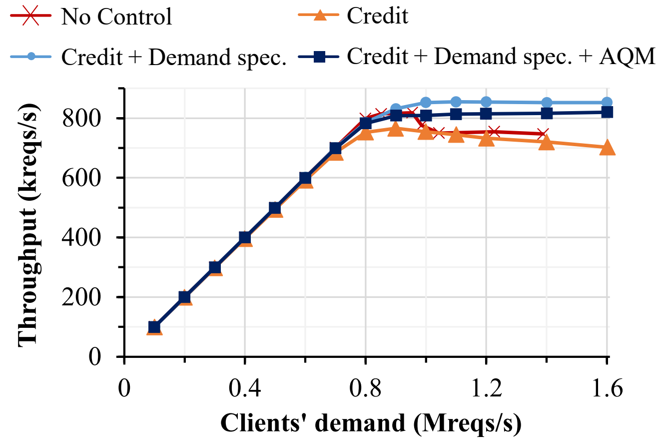
(c) Drop Rate

Figure 2.12: Goodput, message overhead, and drop rate with demand speculation and demand synchronization in Breakwater.

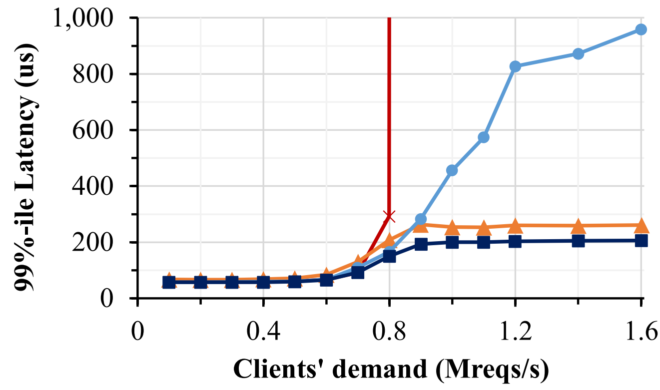
leading to a lower drop rate than demand speculation (Figure 2.12 (c)). Overall, the cost of synchronization between the clients and the server is high in terms of goodput degradation and network overhead, with the small benefit of lowering the drop rate at the server.

Performance Breakdown: To quantify the contribution of each component of Breakwater to its overall performance, we measure the throughput and 99%-ile latency after incrementally activating its three major components: credit-based admission control, demand speculation, and delay-based AQM. The results are shown in Figure 2.13. We use the synthetic workload whose service time is exponentially distributed with 10μs average (SLO = 200 μs). With no overload control at all, throughput starts to degrade, and tail latency soars, making almost all requests violate their SLO as demand becomes higher than server capacity. Credit-based admission control effectively lowers and bounds the tail latency, but throughput still suffers due to the messaging overhead. Demand speculation with message piggybacking reduces the messaging overhead, but it worsens tail latency due to incast caused by credit overcommitment. By employing delay-based AQM, Breakwater effectively handles incast, leading to high throughput and low tail latency.

Parameter Sensitivity: Breakwater parameters are set aggressively to maximize the goodput, resulting in a relatively high drop rate. With less aggressive parameters, Breakwater can drop fewer requests sacrificing goodput. Figure 2.14 demonstrates the trade-off between the goodput and the drop rate for the workload with exponential service time distribution with 10μs average with 1M reqs/s demand from 1,000 clients. The values of pairs of α and β are aligned in descending order of aggressiveness over the x-axis. Breakwater achieves 0.7% of drop rate by sacrificing 2.2% of goodput (with $\alpha = 0.1\%$, $\beta = 8\%$) and 0.4% of drop rate by sacrificing 5.1% of goodput



(a) Throughput



(b) 99%-ile Latency

Figure 2.13: Breakwater performance breakdown.

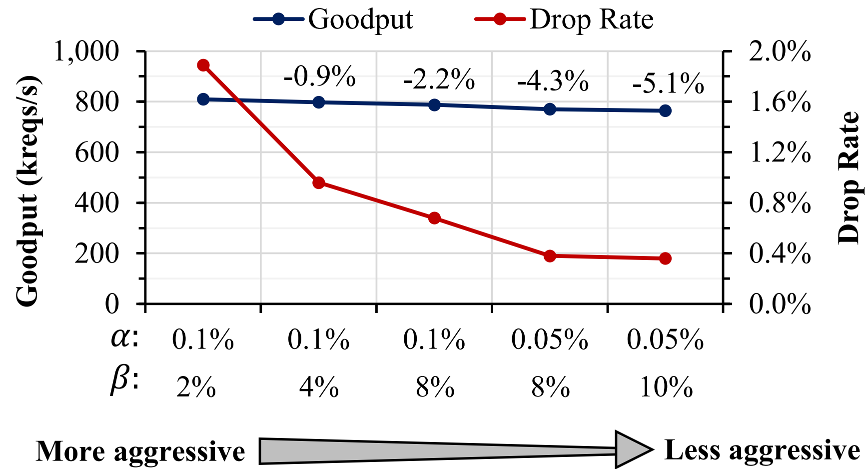


Figure 2.14: Goodput and drop rate with different aggressive parameters of Breakwater.

(with $\alpha = 0.05\%$, $\beta = 10\%$).

In practice, it is not easy to find the best parameter configuration for an operational system. It is even more difficult when traffic patterns change over time because parameter adjustments could be required to achieve the best possible performance. Thus, it is desirable to develop systems that are robust to parameter misconfiguration and changes in traffic patterns, providing consistently good performance even with small errors in parameter settings. Breakwater is robust. In particular, it provides high throughput and low tail latency despite parameter misconfiguration. We compare it against DAGOR and SEDA, measuring their performance for the same workload while varying their parameters. Specifically, we measure the throughput and 99%-ile latency after reconfiguring the three most sensitive parameters for each system: target delay, increment factor, and decrement factor (d_t, α, β for Breakwater; threshold of the average queueing time, α, β

for DAGOR; and $target, adj_i, adj_d$ for SEDA). Given the set of parameters producing best goodput, we measure 27 data points with -10, 0, +10 μs of target queueing delay, 0.5×, 1×, 2× of the increment factor, and 0.5×, 1×, 2× of the decrement factor. We use a synthetic workload with exponentially distributed service times with 10μs average with 1,000 clients. The results are shown in Figure 2.15 where the circles filled with light color indicate the performance with the parameters tuned for the best goodput. All configurations of Breakwater achieve comparable performance in terms of both throughput and tail latency, achieving better throughput and latency trace-offs and more consistent performance with different sets of parameters. DAGOR tends to provide high throughput, but its tail latency is as high as four times the SLO in the worst case. SEDA's worst case tail latency is lower than DAGOR, but it suffers from severe throughput degradation when its parameters are too conservative.

Performance under Realistic Workload

To evaluate Breakwater in a more realistic scenario, we create a scenario where one memcached instance serves 10,000 clients. We use the USR workload from [22] where 99.8% of the requests are GET, and other 0.2% are SET. Each client generates the load according to an open-loop Poisson process. We set an SLO of 50μs considering that the latency of GET operation of memcached is less than 1μs. Figure 2.16 shows goodput, median latency, 99%-ile latency, and drop rate of Breakwater, DAGOR, and SEDA. Breakwater achieves steady goodput, low latency, and low drop rate, whereas both DAGOR and SEDA suffer from goodput degradation with high tail latency caused by incast when the server becomes overloaded. With clients' demand of 2× capacity, Breakwater achieves 5% more goodput and 1.8× lower 99%-

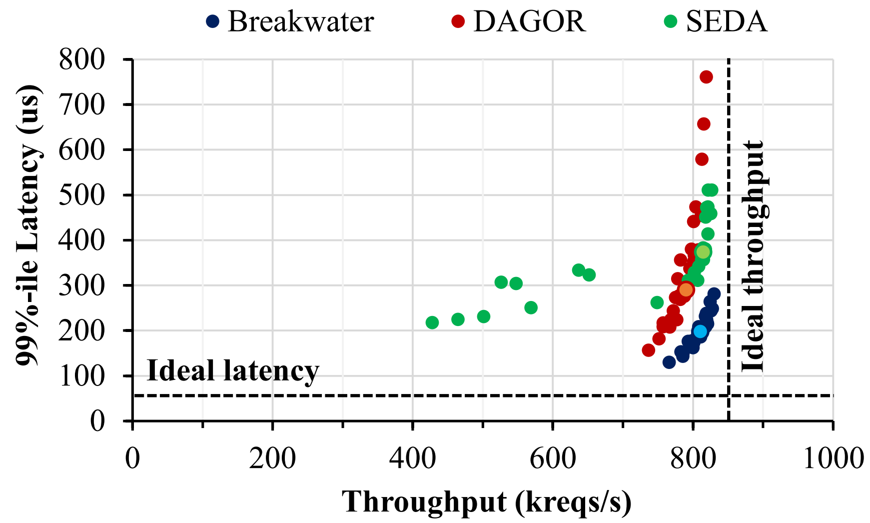


Figure 2.15: Throughput and 99%-ile latency trade-off with different sets of parameters (circle with light color indicates the point producing best goodput).

ile latency than SEDA; and 14.3% more goodput and 2.9× lower 99%-ile latency than DAGOR. Because of bimodally distributed service time with a mix of GET and SET requests, Breakwater shows around 25μs higher 99%-ile latency than its SLO and about 1.5% point higher drop rate than DAGOR.

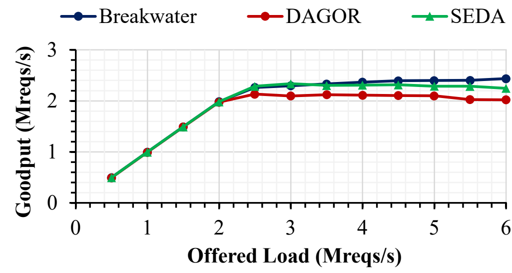
2.5.3 Lock-bottlenecked Scenarios

In this subsection, we examine the scenarios where the locks are the bottleneck of the application. With unpredictable lock contention, we use performance-driven efficiency as the congestion signal to determine whether the server is overloaded or not.

Mutex-intensive Application: Lucene

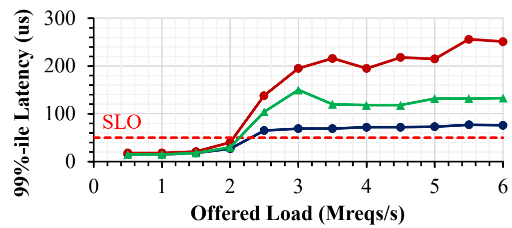
Lock contention inside Lucene: Lucene is a search engine library that maintains two main types of structures: 1) inverted indices, called `Segments`, and 2) per-term scores of all indexed documents, called `TermDocs`. Every `Segment` and `TermDocs` is protected by its own mutex. Every request performs a binary search over all `Segments` to find the documents corresponding to its search query. Then, documents are ranked based on the information found in the `TermDocs` corresponding to the identified documents.

As load increases on the server, the per-`Segment` lock becomes contended because every request needs to search over all the `Segments`. `Segments` containing more entries are more likely to be contended because it takes more time to perform a binary search over their entries. Further, if a specific document becomes popular, the per-`TermDocs` lock protecting its data becomes contended.

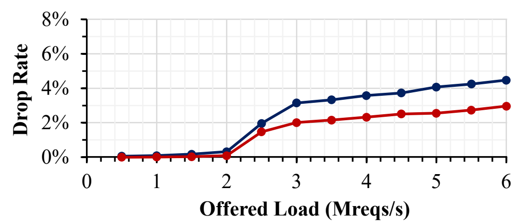


(a) Goodput

(b) Median Latency



(c) 99%-ile Latency



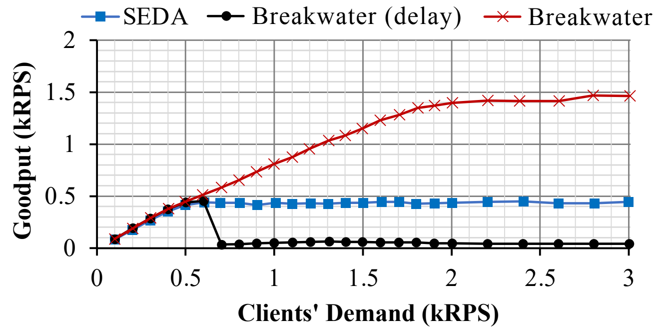
(d) Drop Rate

Figure 2.16: Memcached performance for USR workload with 10,000 clients (SLO = 50 μs).

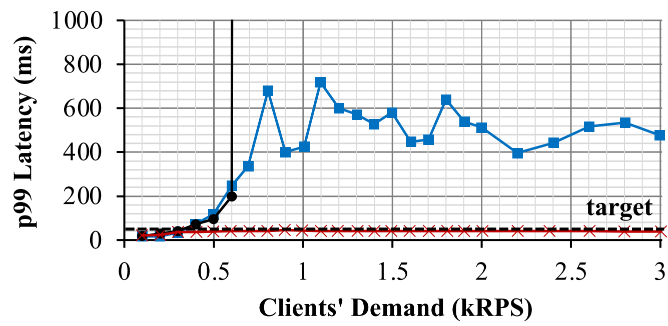
Application modification: We ported the C++ version of Lucene, Lucene++ [108], to Shenango and built a simple in-memory search application, where all the data is stored in memory with `RAMDirectory`. We replaced the per-`Segment` lock and per-`TermDocs` lock with Breakwater's latency-aware synchronization API to allow request drops. In total, we modified 40 LOC of Lucene++ after porting it to Shenango. Note that, while Lucene allows for reporting partial search results, we don't allow that to provide a fair comparison between overload control schemes that don't drop requests. The response contains either the complete search result or a failure notification.

Workload and configuration: We populate the server with a dataset of 403,619 COVID-19-related tweets [79] in English posted between 27th and 29th November 2021. The clients generate single-term search queries. The search term (or word) is sampled from the word distribution in the data set excluding stop words like "a", "the", "and", etc. All the tweets are loaded to the server before serving clients, and tweets are not modified or deleted during an experiment. This workload yields an average processing time of 1.7 ms and a 99th percentile latency of 20 ms on a lightly-loaded server. Thus, we set the target delay to 40 ms. For SEDA, we set $timeout = 1$ s, $adj_i = 0.1$, and $adj_d = 1.3$.

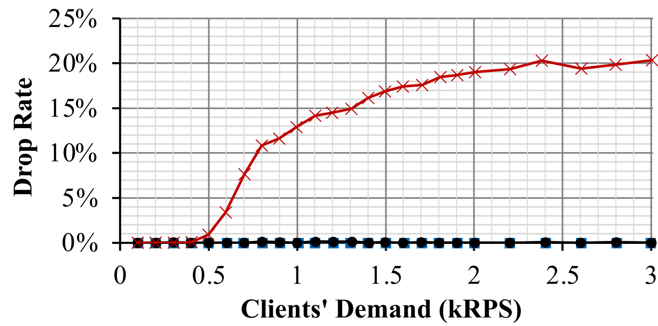
Overall performance: Figure 2.17 shows the goodput, 99th percentile latency, and drop rate for SEDA and Breakwater. Note that Lucene does not suffer from any CPU congestion. Thus, overload control mechanism with CPU-based overload signal such as Breakwater with request queueing delay overload signal never controls the incoming load, leading to congestion collapse as mutexes become congested with clients' demand exceeding 600 RPS. SEDA reduces clients' request sending rate as soon as it measures high latency due to a mutex congestion, reacting to the most congested



(a) Goodput



(b) 99%-ile Latency



(c) Drop Rate

Figure 2.17: Performance of SEDA, Breakwater with request queueing delay overload signal, and Breakwater with efficiency overload signal for Lucene.

data path, which limits the system's goodput to 500 RPS. SEDA's tail latency is bounded but more than 10 times higher than the target latency because of incast. By better utilizing uncongested data paths and dropping the excess load, Breakwater with efficiency overload signal achieves up to 3.3 times higher goodput and 17 times lower 99th percentile latency than SEDA.

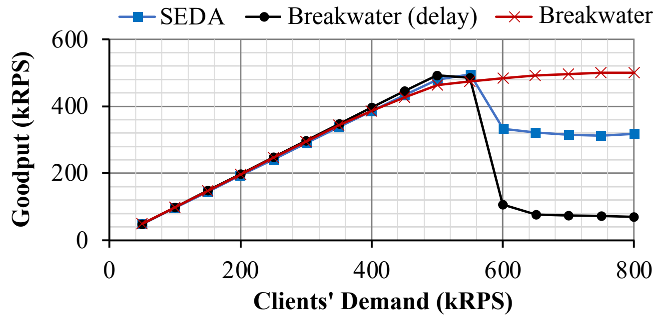
Latency-critical Application: Memcached

Lock contention inside Memcached: The key-value pairs are stored in a giant hash table, composed of multiple hash buckets. Memcached has two main types of locks that may be contended. First, each hash bucket is protected by a mutex called `item_lock`, and this mutex may get contended not only by concurrent accesses (i.e., reads or rights) to the same key but also by accesses on different keys sharing the same key hash. Thus, it's difficult to predict which `item_lock` a request will need before executing it. Second, Memcached manages its memory by assigning items memory from a global pool, which is protected by a global lock called `slabs_lock`. Every `SET` and `UPDATE` request must grab the `slabs_lock` to allocate memory for the new value.

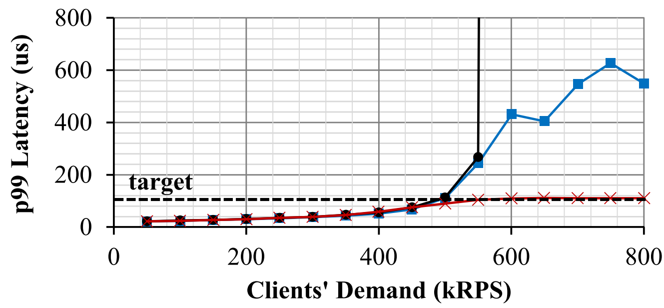
Application modification: We replaced the `item_locks` and `slabs_lock` with Breakwater's latency-aware mutexes. When a request is dropped, Breakwater delivers a failure message to the client immediately. Furthermore, it cleans up the intermediate state processed by the request, freeing up the chunk allocated to the request before the thread handling that request exits. We don't allow drop when a request tries to reacquire `slabs_lock` to free up the memory to avoid memory leaks. In total, we modified 50 LOC in Memcached [110], excluding the modifications to port it to Shenango.

Workload and configuration: For Memcached experiments, we use the VAR workload from Facebook Memcached cluster [30]. VAR is a `SET`-heavy workload for server-side browser information where 82% of the requests are `SET` requests. The key distribution of the workload is skewed with 10% of the keys used by 90% of the requests. With a `SET`-heavy workload, `slabs_lock` becomes the bottleneck as all SET requests require `slabs_lock` to allocate memory region. We approximately follow the key and value size distribution for each workload as described in [30]. We generate 100,000 key-value pairs and use the hash power of 17, providing 131,072 buckets in the hash table, which is sufficient to avoid severe hash collisions. Since `SET` requests complete within less than 1 μs on average, we set the target delay to 110 μs. For SEDA, we set *timeout* to 1 ms, adj_i to 100, and adj_d to 1.02. For Breakwater, we set the initial queueing delay budget to 70 μs.

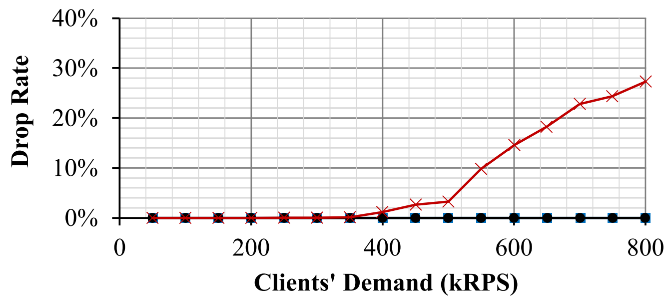
Performance with a global mutex bottleneck: Figure 2.18 demonstrates the performance of the three overload control schemes. When the `slabs_lock` becomes contended with clients' demand of more than 550 kRPS, both Breakwater with delay signal and SEDA experience a goodput drop because of the increase in latency. As with Lucene, the admission control of Breakwater delay signal are not triggered because the CPU is not congested. On the other hand, SEDA suffers from incast. The goodput of Breakwater with efficiency signal increases further by utilizing uncongested data paths with `GET` requests achieving 1.6 times higher goodput than SEDA and 7 times higher goodput than Breakwater. The increment in Breakwater's goodput is limited by the overhead of request drops. Most of the dropped requests are `SET` requests, and some of them require the `slabs_lock` to free the allocated memory. As more requests are dropped, the `slabs_lock` becomes more contended by new `SET` requests that need to allocate the memory as well as old and dropped requests that need to



(a) Goodput



(b) 99%-ile Latency



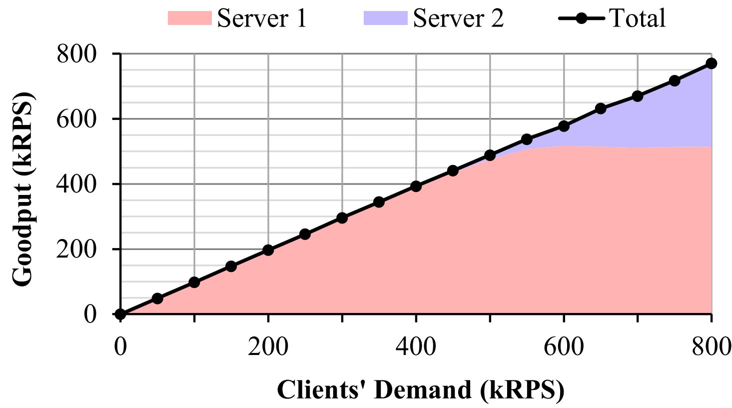
(c) Drop Rate

Figure 2.18: Performance of SEDA, Breakwater with request queueing delay overload signal, and Breakwater with efficiency overload signal for Memcached with VAR workload.

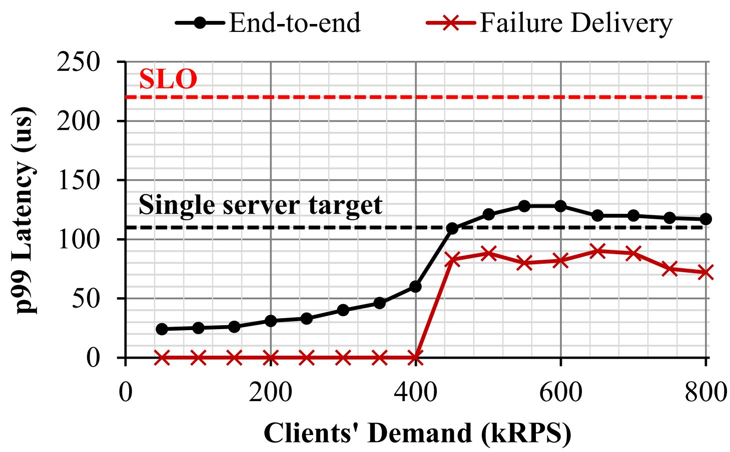
release their memory, resulting in lower throughput of `SET` requests at very high loads.

Maintaining the SLO under retransmissions: To better understand the impact of request drops on the overall SLO, we construct a simple scenario where Memcached has two replicas, but we otherwise use the same configuration as before. When a client makes a request, it sends the request to Server 1. If it is dropped, the client then retransmits it to Server 2 (after receiving a failure message from Server 1). This structure is similar to how Memcached is operated at Facebook [28] where they don't provide a strong consistency guarantee. Note that if both servers are overloaded, the problem ceases to be an overload control problem as the service operator needs to allocate more servers. Thus, our experiment captures the case where there is sufficient capacity to handle all requests, but retransmission may still be necessary. We anticipate up to one retransmission could happen, considering the capacity of the two servers and the demand the clients generate during the experiment, so we set the service-level objective (SLO) to two times the single server target delay, or 220 μs.

Figure 2.19 demonstrates the total goodput of both servers, the 99th percentile end-to-end latency, and failure message delay for the VAR workload. When the clients' demand exceeds 400 kRPS, Server 1 starts to drop requests. Breakwater drops the requests before they wait for the contended mutex if the delay at the mutex exceeds a request's budget. Thus, most of the failure messages are delivered within the target delay. Note that if a client doesn't receive a credit for a request within 10 μs from Server 1, it sends the request to Server 2 with the locally generated failure message. As clients' demand increases, the 99th percentile delay of failure messages decreases because more requests are retransmitted to Server 2 with local failure message. The overall 99th percentile end-to-end latency achieved by



(a) Goodput



(b) 99%-ile Latency

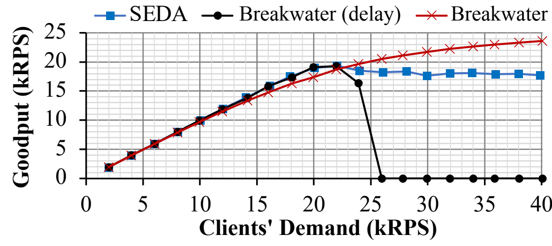
Figure 2.19: Service-level performance of Breakwater for the Memcached VAR workload with retransmission.

Breakwater is higher than the per-server target delay because some requests need to be retransmitted. However, it is still 1.7 $\times$ lower than the SLO.

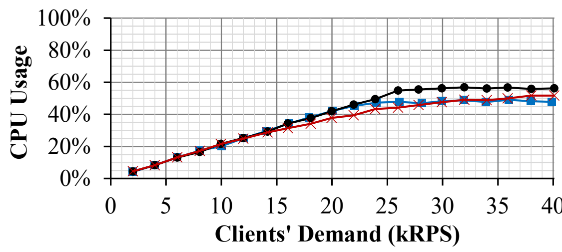
Microbenchmark

Workload and configuration: To further analyze Breakwater's performance, we run the synthetic application depicted in Figure 2.2. We choose the configuration $p = 50\%$, making both data paths equally likely to be congested, to provide a best-case scenario for SEDA. We use a workload with exponential service time distribution of 10 μs. The target delay values are 200 μs. For SEDA, we set $timeout = 1$ ms, $adj_i = 10$, and $adj_d = 1.04$. For Breakwater, we set the initial queueing delay budget to 134 μs and 85 μs, respectively, for the two settings.

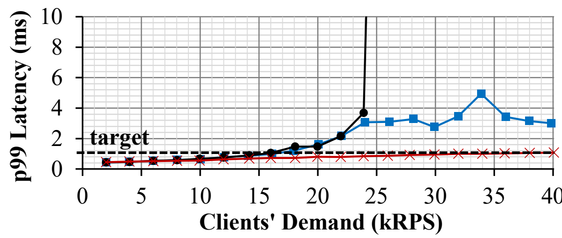
Overall performance: Figure 2.20 shows the goodput, CPU usage, 99th percentile latency, and drop rate for a workload with 10 μs average service time. The performance is bottlenecked by the mutexes, leaving the CPU underutilized even with a high clients' demand. Thus, at high load, the admission control or AQM logic of Breakwater with delay signal is not triggered, leading to congestion collapse. SEDA limits the sending rates of clients as soon as it measures high tail latency with a single temporarily congested data path. Thus, SEDA's goodput is limited to 168 kRPS leaving the other data path uncongested. With a larger clients' demand, SEDA suffers from incast because 1,000 clients are each running a control loop separately. As a result, it shows up to three times higher tail latency than the target delay. Breakwater with efficiency signal improves goodput by up to 32% compared to SEDA, maintaining latency within the target delay by dropping up to 40% of incoming requests. Note that the performance benefits of Breakwater compared to SEDA increase as p deviates from 50%,



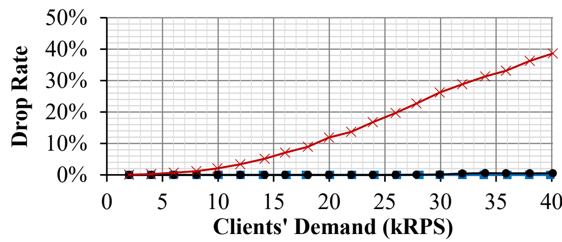
(a) Goodput



(b) CPU Usage



(c) 99%-ile Latency



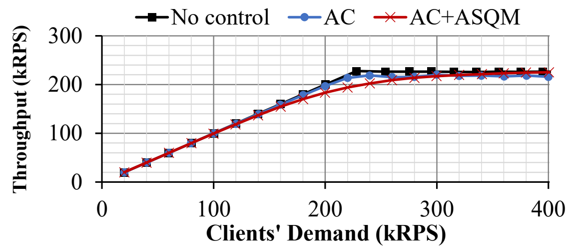
(d) Drop Rate

Figure 2.20: Performance of SEDA, Breakwater with delay signal, and Breakwater with efficiency signal for synthetic workload with $p = 50\%$ and $10\,\mu$s average service time.

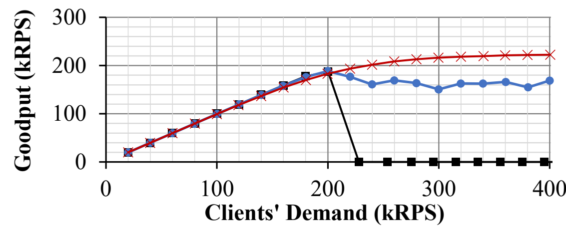
making SEDA more conservative as it reacts to the most congested path.

Performance breakdown: We measure the performance of Breakwater after incrementally activating its two components: the performance-driven admission control scheme (AC) and Active Synchronization Queue Management (ASQM). We run the experiments with the synthetic application with $p = 50\%$ and an average service time of $10\,\mu$s. Figure 2.21 shows the throughput, the goodput, the 99th percentile latency, and drop rate. With no overload control, goodput collapses as soon as one of the data paths becomes congested. Enabling admission control bounds the tail latency by limiting incoming load if there is no throughput improvement. However, when mutexes start to be congested, its goodput degrades with up to three times higher tail latency than the target because one of the mutexes can have a high queueing delay with the requests' probabilistic data path selection. By employing ASQM, Breakwater ensures the tail latency does not miss the target delay by dropping requests.

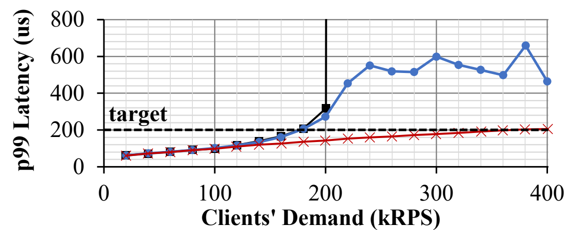
Parameter sensitivity: With efficiency overload signal, Breakwater balances goodput and drop rate using the efficiency threshold (t_e). To quantify the trade-off between them, we repeat the experiment with the synthetic application with $p = 50\%$ and the average service time of $10\,\mu$s varying the t_e from 1% to 50%. Figure 2.22 shows the goodput and drop rate of Breakwater with different t_e values when the clients' demand is 300 kRPS, around 1.4 $\times$ of the capacity (consider Figure 2.21 as a reference). For all values of t_e smaller than 10%, the goodput and drop rate don't change because throughput improvements with a small t_e are always marginal. With larger t_e values, both the goodput and drop rate decrease as admission control targets to operate the server on the left side of the Phase II region in Figure 2.6. With $t_e = 50\%$, it achieves 23% less goodput and 4 $\times$ lower



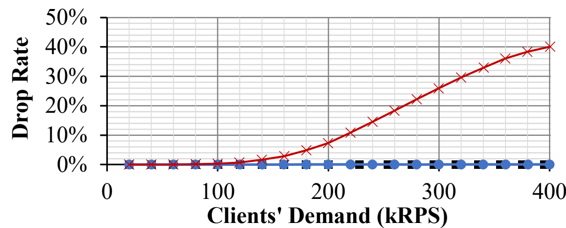
(a) Throughput



(b) Goodput



(c) 99%-ile Latency



(d) Drop Rate

Figure 2.21: Performance of Breakwater by incrementally applying performance-driven admission control (AC) and ASQM with the synthetic application with 10 μs average service time.

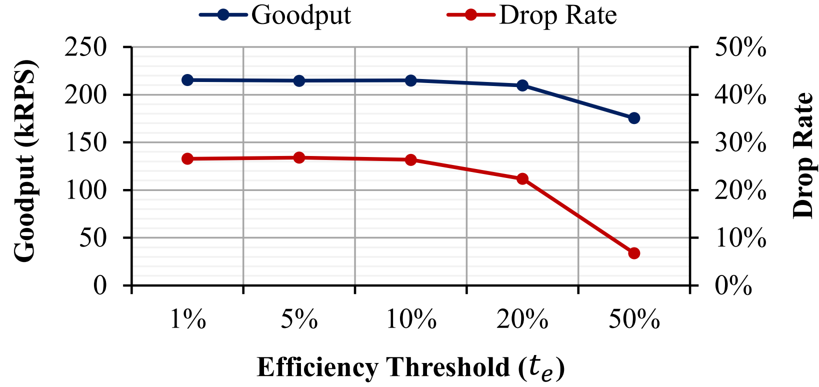


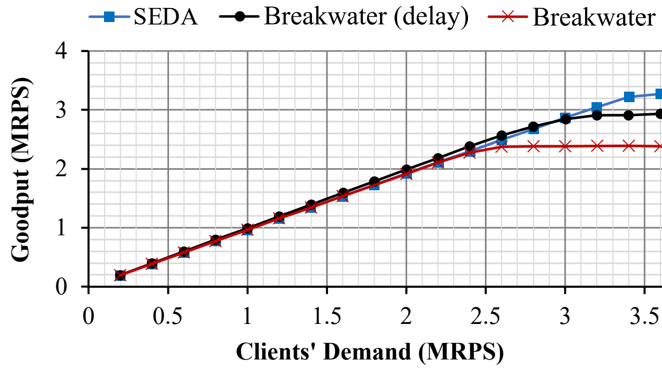
Figure 2.22: Breakwater parameter sensitivity (efficiency threshold, t_e).

drop rate than $t_e = 1\%$, allowing server operators to navigate the tradeoff between goodput and drop rate.

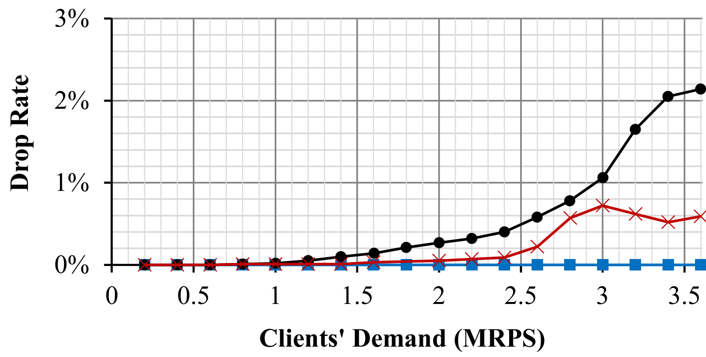
2.5.4 Limitations of Breakwater

Breakwater requires a correct overload signal depending on the bottlenecked resources. Request queueing delay signal works only for CPU congestion while the efficiency signal works for other types of bottleneck including locks. When there is a mismatch between the bottlenecked resources and the overload signal, Breakwater's performance can be severely degraded. In §2.5.3, we demonstrated that lock-bottlenecked applications can experience congestion collapse when Breakwater uses request queueing delay as the overload signal.

To demonstrate the performance of Breakwater with efficiency signal for CPU bottlenecked workload, we repeat the Memcached experiment in §2.5.3 with the USR workload with efficiency overload signal, a `GET`-dominated workload for user account status information where 99.8% of the requests



(a) Goodput



(b) Drop Rate

Figure 2.23: Performance of SEDA, Breakwater with delay signal, and Breakwater with efficiency signal for Memcached with USR workload.

are `GET` requests and about 20% of the keys are used by 80% of the requests. With the USR workload, Memcached saturates the CPU when it's configured with a high enough hash power (i.e., a large number of buckets compared to the number of key-value pairs). However, some `item_locks` can still become congested intermittently because of the skewed key distribution. Figure 2.23 shows the goodput and drop rate, comparing Breakwater with efficiency signal to Breakwater with delay signal and SEDA. With clients' demand of 3.6 million RPS, Breakwater with efficiency signal achieves 37% less goodput than SEDA and 23% less goodput than Breakwater with delay signal.

The USR workload is CPU bottlenecked, allowing CPU-based request queueing delay signal to be effective. Efficiency signal achieves lower goodput than delay signal due to the slow reaction of the admission control. In particular, Breakwater with efficiency signal changes its credit pool size every four end-to-end RTTs. On the other hand, Breakwater with delay signal adjusts its credit pool size every network RTT. As a result, admission control with the efficiency signal reacts to both congestion and added capacity slowly, leading to a lower goodput. SEDA achieves higher goodput than Breakwater because of the overhead of credit management at the server. Specifically, SEDA doesn't add any extra logic at the server while Breakwater performs all their admission control and AQM calculations at the server. This overhead is significant when the request execution time is very small. Note that increasing the number of clients from 1,000 to 10k can lead to performance degradation in SEDA with a larger size of incast [80]. This experiment shows that Breakwater can lead to goodput degradation, especially when there is a mismatch between bottlenecked resources and the overload signal.

2.6 Discussion

Auto-scaling. We do not consider auto-scaling [19, 33, 78] in this chapter, where more resources are provisioned as load increases, as a potential solution for overload control. Auto-scaling can allocate enough capacity over time, but because it operates at the timescale of minutes, it is too slow to resolve microsecond-scale imbalances. Furthermore, over-provisioning resources can be cost-inefficient if used to handle transient spikes in demand, such as those that occur during temporary failures [16].

Fairness. When the server has a sufficient number of credits, it tries to approximate max-min fairness when distributing credits to clients. However, when the number of available credits is less than the number of clients, Breakwater does not provide fairness to clients. Instead, it favors clients for which it is currently processing requests. This allows the server to piggyback credits to the responses and avoid sending explicit credit messages. This preference toward a subset of clients is common in production services [70]. If a service operator wants to provide fairness among clients, the clients receiving the most credits could be timed-out over a longer timescale, so clients starved of credits can get a chance to send instead.

2.7 Related Work

Receiver-driven transport protocols. Homa [66], NDP [55], and ExpressPass [54] schedule network packets with a receiver-driven mechanism to achieve high throughput and low latency. While Homa and Breakwater share some similarities including a credit-based, receiver-driven scheme and credit overcommitment, they are different in three significant aspects. First, Homa handles network congestion, whereas Breakwater handles server over-

load, which means that Breakwater must handle the additional challenges posed by overload control discussed in §2.2.5. Second, Homa relies on full knowledge of clients' demand, whereas Breakwater does not. Instead, the Breakwater server speculates clients' demand based on the latest demand information, the number of clients, and the number of available credits to minimize the message overhead. Third, both the motivation and the mechanism of overcommitment are different. Homa overcommits a fixed number of credits to handle an all-to-all workload, where a sender may get credits from multiple receivers and therefore not be able to send to all of them simultaneously. In Breakwater, however, the server does not know which clients have demand. Thus, it dynamically increases the amount of overcommitted credits until it receives sufficient requests to keep itself busy with demand speculation.

Transport protocol for μs-scale RPCs. R2P2 [74] is a request/response-aware transport protocol designed for μs-scale RPCs. It implements JBSQ inside a programmable switch to better load balance requests among multiple servers. R2P2 limits the number of requests in a server's queue by explicitly pulling the requests from the switch. Through this mechanism, R2P2 provides bounded request queueing and low tail latency when the clients' demand is less than the servers' capacity. However, R2P2 does not provide any server overload control mechanism. If the clients' demand exceeds the servers' capacity, the request queue will build up at the switch, causing requests to violate their SLO. SVEN [83] builds upon R2P2 by adding a server overload control mechanism. Specifically, it drops requests at the switch if sampled tail latency exceeds an SLO-derived threshold. SVEN avoids the cost of request drops at the server by dropping requests early at the switch. However, unlike Breakwater, message overhead increases as clients' demand increases.

Circuit breaker in proxy. Envoy [81], HAProxy [82], NGINX [15], and GateKeeper [12] provide circuit breaker mechanisms to prevent back-end server overload. These proxies sit in front of a back-end server and stop forwarding requests to the server when one of the load metrics (e.g., the number of connections, the number of outstanding requests, the response time, estimated load) exceeds a threshold. However, since those thresholds must be set manually, it's challenging to find the right threshold value that maximizes resource utilization while keeping latency low.

Server overload control. Session-based admission control [6, 7] prevents web server overloads by limiting the creation of new sessions based on the number of successfully completed sessions or QoS metrics. However, they are not compatible with request-response models as they cannot prevent server overloads caused by a single session from a proxy that forwards requests from multiple clients. CoDel [24] controls the queuing delay of a server to prevent server overloads. Still, if the incoming packet rate is high and the CPU is used more for packet processing, the server becomes less CPU efficient and degrades throughput. ORCA [75], SEDA [8], and Doorman [50] rate limit clients so that their sending rates do not exceed the server capacity. Doorman requires manual setting of the server capacity threshold. Both ORCA and SEDA may suffer from long queueing delays or under-utilization if clients make mistakes in their sending rate with stale congestion information from the server. DAGOR [70] takes a hybrid approach using both AQM and client-side rate limiting using adaptive parameter based on queueing delay. However, as DAGOR server updates the congestion status with responses, clients still can undershoot or overshoot the server capacity with stale information on server congestion when client demand is sporadic.

Flow control. TCP flow control prevents the sender from transmitting

more bytes than the receiver can accommodate. The objective of TCP flow control is to avoid memory overrun at the server, not to prevent server overload or SLO violations. More recently, an SLO-aware TCP flow control mechanism [65] was proposed where the server adjusts receive window size in TCP header based on SLO and the queueing delay at the server. This approach limits the "bytes" of the incoming requests to prevent server overload, but it's challenging to decide the appropriate receive window size, especially when the request size is variable.

Measurement-based network congestion control. BBR [49] and PCC [39] employ mechanisms similar to Breakwater's performance-driven admission control. BBR explores the maximum network bandwidth by measuring the throughput with increasing window size. It concludes that the network bandwidth has reached its maximum value if it observes less than 25% of bandwidth increase with doubled window size. Unlike Breakwater, BBR does not utilize a performance-based approach to detect network congestion but to determine a parameter used for congestion control. In PCC, the system operator defines a utility function (e.g., TCP friendliness, latency, or throughput). PCC conducts multiple micro-experiments with a randomized set of parameters to find the configuration that achieves the highest utility. PCC-like algorithms require multiple rounds to find the best configuration, which slows down the reaction of the algorithm to the congestion. Unlike PCC, Breakwater deterministically modifies the credit pool size based on the measurement, which makes its reaction to congestion faster.

2.8 Conclusion

This chapter describes Breakwater, a server-driven, credit-based overload control system for microsecond-scale RPCs. Breakwater achieves high throughput and low latency regardless of the RPC service time, the load at the server, and the number of clients generating the load. Breakwater generates credits based on queueing delay at the server, maintaining high utilization by targeting non-zero queueing delay while avoiding queue buildup. To minimize the overhead of coordination between the clients and the server, we propose demand speculation and credit overcommitment to realize the credit-based design for overload control with minimal overhead. By estimating clients' demand and issuing more credits than their capacity, Breakwater eliminates the extra messaging cost which is often required with a credit-based approach. Additionally, Breakwater reduces its remaining messaging overhead significantly by piggybacking demands and credits to requests and responses, respectively. Our evaluation of Breakwater shows that it outperforms state-of-the-art overload control systems under CPU-bottlenecked and lock-bottlenecked scenarios. In particular, with CPU bottleneck, Breakwater achieves 25× faster convergence with 6% higher converged goodput than DAGOR and 79× faster convergence with 3% higher converged goodput than SEDA when the clients' demand suddenly spikes to 1.4× capacity; with lock contention, Breakwater achieves up to 3.3× higher goodput with 12.2× lower 99th percentile latency than SEDA when applied to Lucene, a realistic search workload.

3 LDB: An Efficient Latency Profiling Tool for Multithreaded Applications

3.1 Introduction

Modern datacenter services like search, social networks, and DNN training operate on huge datasets with complex communication patterns and large numbers of servers [9, 64]. Tail latency is a key challenge in this setting because overall performance is often limited by the slowest response [27]. Despite the tremendous effort that goes into optimizing latency-sensitive programs, operators tend to treat high tail latency as inevitable due to the complexity of deployed programs. Therefore, the main method available to operators today is to keep machine utilization low to control for tail latency, wasting both power efficiency and money [14].

In this chapter, our aim is to empower developers to tackle tail latency problems head-on by answering the following question: *Can a debugging tool identify the precise source of tail latency experienced by a request in a server (e.g., the line of code that is responsible)?* This is a significant challenge, as the effort needed to understand tail behaviors is formidable

with the tools that exist today. Statistical profilers (e.g., Linux's perf-tool), for example, have only limited utility because their method of periodic sampling captures the average runtime of functions, which may deviate significantly from the tail runtime. Further, they don't account for request semantics, so they cannot differentiate between requests running on the critical path versus the background, making it hard to identify bottlenecks. Instead, developers commonly hand instrument code locations that they *suspect* are problematic, but they can only try a few locations at a time due to instrumentation overhead. Thus, a typical workflow involves multiple iterations of instrumentation location adjustment, deployment, and data collection.

One way to avoid this tedious process would be to use a tool that can instrument all functions at a time (e.g., XRay). However, this approach causes significant overhead that can distort an application's behavior. A less invasive option would be to use hardware assistance. For example, Intel recently introduced a CPU extension called Intel Processor Trace (Intel PT) that records every control flow operation (calls, branches, jumps, etc.) to an in-memory log for analysis. NSight recently demonstrated that Intel PT can be used to derive rich tail latency insights, such as a precise timeline of how cycles are spent handling network requests [91].

Unfortunately, Intel PT has drawbacks that make it difficult to use for profiling latency in practice. First, Intel PT is proprietary and requires hardware support, so it is only available on certain platforms. Second, Intel PT generates data at an enormous rate up to more than 1 GB/s, so it is only feasible to record a few seconds of samples. Finally, Intel PT's compression scheme requires a software decoder that walks a program's object code to reconstruct its control flow. This requires several hours of processing—even for a few seconds of data—prohibiting interactive profiling (§3.6.3).

We present LDB, a new latency debugging tool that provides unprecedented visibility into the latency behavior of applications. LDB reports the distribution of the latency of all functions in a process. Furthermore, it allows developers to breakdown the latency faced by a specific request, even when processed by multiple threads, allowing them to zoom in and identify the code responsible for anomalous behavior. LDB provides this information after only seconds of decoding and without significantly harming the performance of the profiled program, enabling monitoring in production environments. In contrast to Intel PT, LDB is also hardware agnostic. In principle, it can be ported to any architecture, and we demonstrate its use on Intel and AMD processors.

The efficiency and portability of LDB stem from a novel, software-only technique, called *stack sampling*. Unlike prior approaches, stack sampling doesn't record timestamps from within application threads (e.g., as obtained by the `RDTSC` instruction) to an in-memory log, so it is much lighter weight ($<$ 0.5 ns per function call) [47, 69]. Instead, a separate stack scanner thread polls the stack of every application thread. During each polling cycle, the stack scanner thread performs a backtrace on each stack to inspect changes to call frames. Intuitively, a function's call frame will be resident on the stack until the function returns, so the more it is responsible for latency, the longer its call frame will remain resident. LDB exploits this to capture the runtime of all the functions that contribute meaningfully to latency (i.e., those that last longer than its sub-us polling interval). As the tail behavior involves longer function execution times, LDB is well-equipped to identify and analyze it.

While stack sampling is based on a simple premise, we had to overcome several challenges to make it work in practice. First, it is not possible for one core to access another core's stack pointer register, so we had to find an

alternative way to locate the deepest call frame. Second, there is not enough information available in call frames to discern between repeated invocations of the same function so we had to find a way to detect them. Third, the stack scanning thread could race with application threads causing it to observe corrupted call frames, so had to develop a mechanism to detect and discard bad samples. Finally, backtracing can cause false sharing with variables on the stack, negatively impacting application performance, so we needed a way to limit this overhead without sacrificing resolution. We discuss our solution to each of these problems in §3.3.2.

In addition to efficiency improvements, LDB provides better visibility into latency problems through *event tagging*, recording several types of events with timestamps and event-specific metadata. Examples include the start and end of requests; cross-thread interactions like locks; and the transfer of request ownership among threads. This allows LDB to track the timeline of each request and correlate this information across multiple threads. For example, LDB can identify a slow function running inside a critical section that is protected by a lock, and then tie it back to a request that is blocked in another thread waiting to acquire the same lock. LDB also uses event tagging to track context switching, allowing it to differentiate between delays caused by the OS scheduler and the application itself.

We demonstrate the value of LDB by profiling two latency-sensitive applications (Memcached and Lucene) and a best-effort application, Qperf, a throughput benchmarking tool for Fastly's implementation of the QUIC transport. We show that LDB allows for detecting complex interactions between threads and identifies functions that significantly impact performance (latency and throughput). Then, we provide a thorough evaluation of the performance of LDB when used to profile the three applications presented in the use cases. In particular, we show that the overhead of LDB is less

than the overhead of Coz and Xray; and comparable to the one of Intel PT on the latest Intel architecture (Ice Lake). LDB maintains its low overhead across Intel and AMD architectures. On the other hand, the overhead of Intel PT increases considerably when used on older Intel architectures.

LDB has some limitations. First, it cannot capture some OS or hardware events like traps or interrupts. Second, LDB requires programs or libraries to be recompiled to support stack sampling, so it cannot trace function latency for unmodified binaries or libraries. Finally, to get the best possible visibility, LDB requires request annotations in the source code (typically just a few lines), but it can still provide useful information including statistics of latencies for each function without annotations.

3.2 Motivation

3.2.1 Debugging the Tail Latency

Consider the example shown in Figure 3.1, based on a pattern found in many real programs. A request processing and background thread require synchronized access to the same data. The request processing thread normally responds with low latency by executing `request_handler()`, which adds items to a `std::map`. Concurrently, the background thread takes a snapshot of the `std::map` every 10ms. Access to the `std::map` is serialized through a `std::mutex`. Figure 3.2 shows a CCDF of the latency of the `request_handler()` function. At the tail, its latency jumps from 1 us to 10 ms (a 10,000× increase)!

This is a challenging issue to debug because it is caused by a rare interaction across two threads. LDB, however, can easily identify the root cause. It captures everything that happened in the program and can gen-

```
1  std::mutex lock;
2  std::map<int, std::string> db;
3
4  void snapshot() {
5    std::ofstream out("snapshot.txt");
6    std::lock_guard<std::mutex> g(lock);
7    for (const auto& kv : db)
8      out << kv.first << "," << kv.second << std::endl;
9    out.close();
10 }
11
12 void background_thread() {
13   while (true) {
14     snapshot();
15     usleep(10000);
16   }
17 }
18
19 void request_handler(int key, std::string& value) {
20   std::lock_guard<std::mutex> g(lock);
21   db[key] = value;
22 }
23
24 int main() {
25   std::thread bg_thread(background_thread);
26   for (int i = 0; i < kRounds; i++) {
27     int key = std::rand() % dbSize;
28     std::string value = generate_random_string();
29     request_handler(key, value);
30   }
31 }
```

Figure 3.1: example.cc: a simple multithreaded program where a foreground thread handles user requests and a background thread snapshots program state every 10 ms.

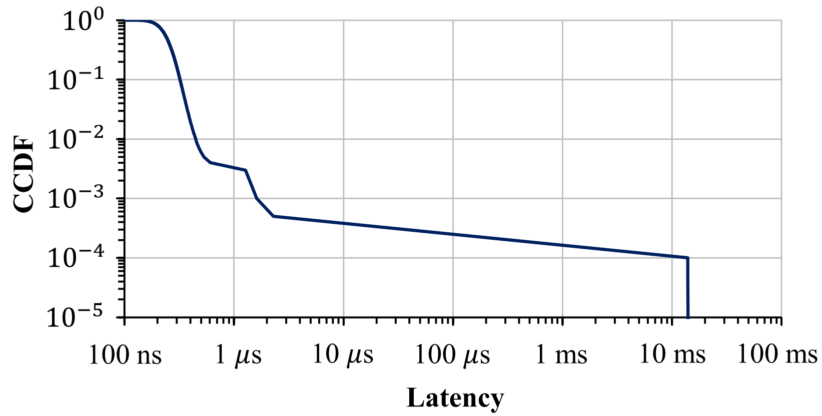


Figure 3.2: Latency distribution of `request_handler()` in Fig 3.1.

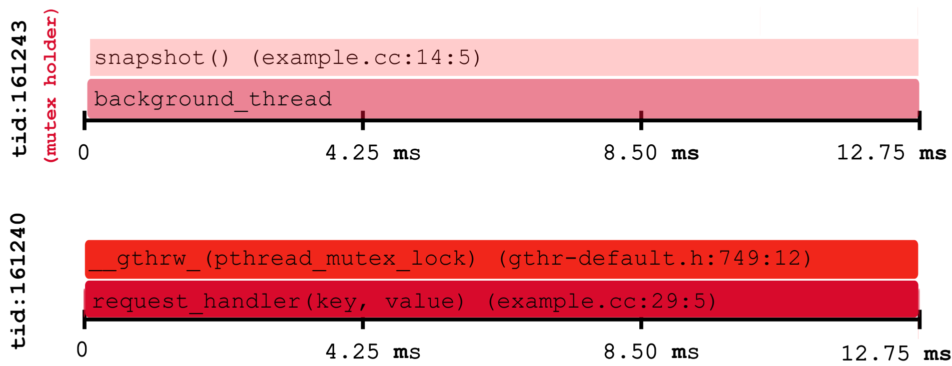


Figure 3.3: A timeline visualization of the time spent in each function during the longest request. The thread on the top is the mutex holder, while the thread on the bottom is the request handler, which is blocked waiting to acquire the mutex.

erate a timeline visualization for each request that includes all the threads that were involved. By plotting the longest request—an output generated by LDB—as shown in Figure 3.3, it becomes clear that the snapshot thread (shown on top) delayed `request_handler()` (shown on bottom) by holding the mutex it was trying to acquire. This suggests that tail latency can be improved by optimizing `snapshot()` or reducing the size of its critical section.

However, existing profilers struggle to debug tail latency issues like these. For example, Figure 3.4 shows the output of perf, one of the widely used debugging tools. The majority of time is spent in `generate_random_string()` and other functions under `request_handler()`. `snapshot()` accounts for only 0.6% and was buried under other 13 miscellaneous functions. This result reveals three interesting problems of using perf for tail latency debugging. First, tail behavior is amortized, so it gets buried down under average behaviors. Second, perf is measuring where the CPU cycles go, not how long each function takes, so it is unable to show the time spent on blocking I/O or synchronization. Figure 3.3 and Figure 3.2 suggest that `snapshot()` runs for over 10 ms and then sleeps for 10 ms, so it should account for at least about 50% time on average. However, much of the time spent on `snapshot()` is spent blocking on I/O, so perf reports only 0.62%. Lastly, perf cannot capture the interplay across threads caused by the mutex.

Intuition. We pay attention to the observation that the metadata in x86 stack frames (e.g., the number of stack frames, return instruction pointers, saved based pointers, etc.) remains unchanged as long as a thread is executing a bottlenecked function. LDB takes an approach in which a separate dedicated busy-running thread, stack scanning thread, periodically scans these stack frames. It then measures the latency of the function call by

Function	CPU Time ▽
`generate_random_string`	63.75%
`request_handler`	7.43%
`std::_Rb_tree_increment`	2.82%
...(13 more functions)...	
`snapshot`	0.62%

Figure 3.4: Perf's output with the example application.

examining whether the metadata in the stack frame metadata stays consistent. If a change is detected in this metadata, it signifies that a function has either returned or that a new function call has been invoked.

3.2.2 Challenges

To realize this *stack sampling* idea entails the following challenges:

1. Finding out the most recent stack frame. In the architecture of x86 stack frames, the stack frames form a singly linked list data structure. Starting with the most recent stack frame, one can traverse the entire call stack by following the saved base pointers. This traversal is necessary for LDB to ascertain whether the stack frame metadata has been modified or not. The location of the most recent stack frame can be retrieved from the RBP register. However, threads other than the application thread itself cannot access this register, making it challenging for the stack scanning thread to determine where to commence the traversal of the stack frames.

2. Differentiating stack frames for different function calls. When function calls are invoked within the same line of code (such as within a loop), they may have identical metadata in their respective stack frames. This can lead to confusion in the function call latency measurement performed by

the stack scanning thread, as it may fail to detect any changes even when a function has returned and a new function call has been invoked. To accurately measure the latency of individual function calls, it is necessary to find a way to differentiate between stack frames from distinct function calls, even if their stack frame metadata appears the same.

3. Cache thrashing and false sharing. Stack frames are frequently accessed by the application thread for local variables. If the stack scanning thread accesses stack frames too often, it may lead to performance degradation because of cache thrashing and false sharing. Stack scanning thread's repeated access to stack frames causes the data to be continually invalidated from the application thread's cache, increasing the delay to access the stack frame.

4. Data race for the stack frames. While the stack scanning thread is traversing the stack frame, the stack frame can be concurrently modified by the application thread with a function return or a new function call. This data race for the stack frames can result in the stack scanning thread collecting incorrect data. Consequently, it leads to inaccurate measurement of function latencies. For precise latency measurement, we need a way to detect and gracefully handle such data races, ensuring the integrity of the data collected by stack scanning thread.

3.3 System Design

3.3.1 Overview

Our objective is to create a lightweight, portable latency profiling tool that can capture fine-grained information about the time spent in each function

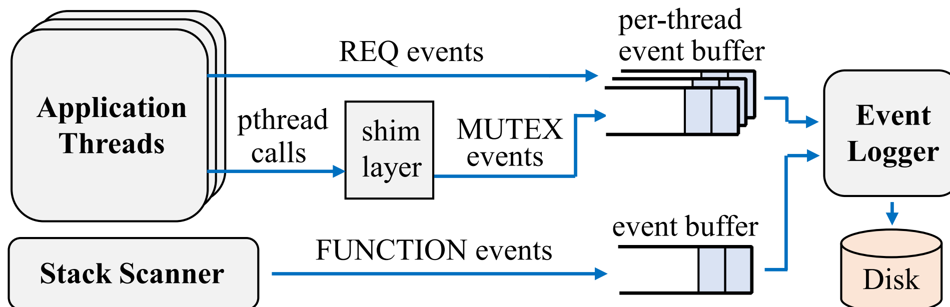


Figure 3.5: The flow of events that are recorded by LDB.

in a program. Thus, the per-function cost has to be minimal. We achieve this through two key ideas. First, we use a separate busy-polling core to shift away the instrumentation cost that would normally be incurred inside program threads, such as capturing timestamps and recording events to an in-memory log. Second, we reduce the trace data generation rate by recording only functions with call frames that are resident on the stack for longer than the polling interval. Intuitively, very short functions do not contribute to latency, so it is okay to not spend resources in capturing them.

Building upon these ideas, we propose a new technique, called stack sampling, where a stack scanner thread repeatedly scans the stacks of application threads. By observing the persistence of specific call frames across multiple scans, the stack scanner thread can estimate each function's invocation latency. These invocation latencies can then be integrated with other event sources (e.g., acquiring a mutex, starting to process a request, spawning a thread, etc.) that are tagged with metadata and synchronized timestamps. This enables greater visibility, such as capturing locking interactions across threads.

Event recording. Figure 3.5 shows how different types of events are

<table>
<tr><th>Event Category</th><th>Event Code</th><th>Trigger Condition</th><th colspan="2">Metadata</th></tr>
<tr><td>Stack samples</td><td>FUNCTION</td><td>A function returned</td><td rowspan="10">- timestamp
- thread ID</td><td>- caller's PC
- latency
- function IP</td></tr>
<tr><td rowspan="4">Request</td><td>REQ_START</td><td>A request started (or resumed from a queue)</td><td>- request ID
(- queue address)</td></tr>
<tr><td>REQ_BLOCK</td><td>A request was inserted into a software queue</td><td>- request ID
- queue address</td></tr>
<tr><td>REQ_END</td><td>A request ended</td><td>- request ID</td></tr>
<tr><td>REQ_END_ALL</td><td>All pending requests ended (on this thread)</td><td></td></tr>
<tr><td rowspan="3">Synchronization</td><td>MUTEX_WAIT</td><td>A thread waited for a mutex</td><td>- mutex address</td></tr>
<tr><td>MUTEX_LOCK</td><td>A thread acquired a mutex</td><td>- mutex address</td></tr>
<tr><td>MUTEX_UNLOCK</td><td>A thread released a mutex</td><td>- mutex address</td></tr>
<tr><td rowspan="2">Scheduling*</td><td>SCHED_SWITCH</td><td>A thread was context switched by the OS</td><td>- next thread ID
- CPU ID</td></tr>
<tr><td>SCHED_MIGRATE</td><td>A thread was migrated to a different core by the OS</td><td>- origin CPU ID
- destination CPU ID</td></tr>
</table>

* Scheduling events are collected by an external tool from OS.

Figure 3.6: The types of events that are tagged and recorded by LDB.

tagged and recorded by LDB. LDB has three main components that generate events. First, a *stack scanner*, which runs in a busy-polling thread, scans application threads' stack and records invocation latencies each time a function returns. Second, a *shim layer* intercepts common threading operations (e.g., `pthread_mutex_lock()`) and records an event before forwarding the operation to its underlying implementation. Finally, application threads can generate events directly when they are annotated by the programmer, such as the start and end of a request. LDB records all events to per-thread shared-memory queues to improve scalability. An *event logger*, running in a separate thread, then gathers the events and stores them to disk for later analysis. Separately, the existing OS performance monitoring subsystem can be used to record scheduling events (not shown) like context switches and thread migrations [94]. Our design is extensible, and we plan to add additional event sources in the future, such as recording delays caused by interrupts. A listing of all the events that LDB tracks is shown in Figure 3.6.

3.3.2 Stack Sampling

Compiler instrumentation. LDB relies on compiler instrumentation that it inserts as small, low-overhead changes to the function calling conventions. First, the compiler emits a frame pointer for each call frame. Normally, most compilers optimize away frame pointers, but they are needed by the stack scanner to backtrace the stack. While functions can be identified using the return address saved on the call frame, this value doesn't allow us to differentiate between multiple invocations of the same function. This difference is critical for latency debugging, as we care about the per-invocation latency on each function, not aggregate measures like CPU time. To resolve this problem, LDB uses generation numbers to differentiate different invocations to the same function. If a generation number is different in an otherwise identical call frame, LDB knows that it was a separate invocation. The compiler appends a generation number to each call frame. The generation number is a monotonically increasing number, derived by incrementing a word stored in thread-local storage (TLS). Finally, the compiler records the frame base pointer of the deepest call frame (i.e., the RBP), also placing it in TLS. The use of TLS avoids cache contention between application threads, allowing LDB to scale well across cores.

Sampling the stack. Figure 3.7 illustrates how the *stack scanner* samples the stacks of application threads. The stack scanner runs as a separate thread in the same process as the application, allowing it to share its address space. The stack scanner maintains a table of the application threads that are currently running (①). For each application thread, it fetches the frame pointer of the deepest call frame, which the compiler stores in TLS, (②) and starts scanning the stack (③). It traverses all the call frames up to the main function by following the stack frame pointers (④). While traversing

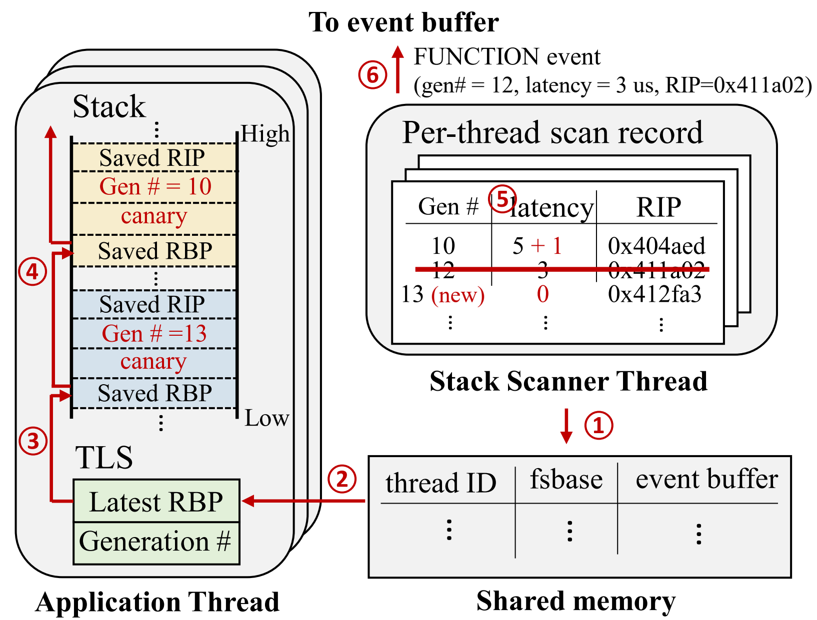


Figure 3.7: Stack Scanner Thread's Stack Sampling.

the stack, it collects each generation number, which is located at a fixed offset from the current frame pointer, along with the return address of the call frame.

Latency calculations. When the stack scanner collects information from the stack, it updates its *scan records*, which are a table of metadata for each call frame (⑤). If the scanner detects a new call frame, it creates a new scan record and records the current timestamp and generation number. If an existing scan record's call frame is not found during the new scan, LDB concludes that the function has returned and generates the FUNCTION event with the generation number, latency, and return address (⑥). It then removes the scan record. We now discuss the various enhancements we made to this basic procedure to address validating call frames, avoiding race conditions, and minimizing probing effects.

Validating call frames. Another challenge is in identifying valid call frames. For example, even a program compiled using LDB's compiler may still be linked against library code that does not contain instrumentation. Thus, some call frames may not have valid generation numbers. To detect this, LDB reserves an additional eight bytes in the call frame, called a canary. The canary contains a known magic value that the stack scanner looks for before parsing the generation number. If it is missing, latency is not reported for that function, but any parent functions that have the canary will still be reported. To avoid pursuing invalid stack frames, LDB stops traversing when the canary in the current stack frame is invalid, or the next stack base pointer is invalid. Thus, LDB is guaranteed to terminate its stack traversal. Further, it avoids segmentation faults by validating whether a certain memory address is between the start of the stack (base pointer of the very first stack frame recorded at thread start) and the end of the stack (latest RBP value in the thread local storage) before reading it.

Preventing data races. The application thread could race with stack scanning if it calls or returns from functions while the stack scanner thread is traversing the stack. To avoid collecting invalid call frames, the stack scanner uses TLS data as a sequential lock, a form of optimistic concurrency control [112]. Because the frame pointer changes with each function call or return, and the generation number changes with each function call, TLS data can be used to verify that the collected generation numbers are valid. The stack scanner compares TLS data before and after each stack scan, and if they don't match, it discards the collected sample and tries again.

Reducing probing effects. Another potential concern is that reading the stack could impact an application's performance. For example, if an application thread frequently modifies a variable stored on its stack, and it lands in the same cache line as a call frame, this could result in false sharing between the stack scanner and the application.

To prevent this, LDB uses TLS data to detect function calls and returns, and initiates stack sampling only when they occur. This avoids all false sharing during function execution. The stack scanner supports this by polling the generation number stored in each TLS region, and waiting for it to increase before sampling the thread's stack. The stack scanner then proceeds with the scan, retrying if there was a race condition (which is rare). Once it gets a valid sample, it stops scanning until the next time the generation number increases. The generation number is placed in a dedicated cache line, allowing it to remain in the shared cache state. Therefore, no coherence traffic is generated while it is polled (until it is modified). LDB also supports pausing the stack scanner between probes (e.g., delaying for $1\,\mu$s). However, we found that the above technique allows LDB to poll in a tight loop with negligible probing effects and better resolution.

3.3.3 Tracing Cross-thread Request Handling

LDB analyzes cross-thread interaction with three types of events: request events, synchronization events, and scheduling events. Each event is timestamped and included in the trace. The time duration between two events (e.g., waiting for and acquiring a lock) along with other functions that happen between the two events, help construct a rich timeline. To minimize the extra latency required for event logging, each event is recorded to a per-thread circular event buffer. Then, the events in the event buffers are polled by the event logger which persists the events to disk.

Request events. For multi-threaded applications, it is hard to figure out which threads are responsible for a long latency. To enable per-request tracing with a multi-thread environment, LDB provides an API for developers to annotate when a thread starts and finishes handling a request. Using request annotations, LDB constructs the timeline for a specific request showing the interaction between the threads handling the same request and revealing which threads contribute to a long request processing time. In particular, all functions invocations in a thread that happens between a REQ_START and a REQ_END are counted towards the timeline of the processing of that request. We rely on the application developers to correctly tag events, including if a request is temporarily placed in a queue (i.e., REQ_BLOCK), in order for LDB to construct an accurate timeline.

Synchronization events. Contention for shared resources can be a major source of latency. Visibility into synchronization events (i.e., mutex wait, mutex acquire, and mutex release) can play a key role in identifying performance bottlenecks in the presence of cross-thread interactions. Mutex events reveal not only which mutex is contended and how long it delays a request but what the mutex holder thread is doing while hold-

ing the mutex. For mutex events recording, LDB interposes pthread calls (`pthread_mutex_lock()` and `pthread_mutex_unlock()`) and generates the MUTEX_WAIT, MUTEX_LOCK, and MUTEX_UNLOCK events. We discovered recording every mutex event can introduce extra overhead, especially for mutex-intensive applications. To minimize this overhead, LDB decides whether it should record mutex events outside of the critical section after releasing the mutex with `pthread_mutex_unlock()`. If either mutex wait time or lock time exceeds MUTEX_EVENT_THRESH (1 μs by default), it records the mutex event in the event buffer.

Scheduling events. The operating system scheduler can contribute to request latency through context switching between applications or threads. Revealing the delay caused by context switches can guide the developers to look at operating system configurations, not the application, to improve latency. Unlike other types of events, LDB collects the scheduling events from an external source. In particular, we collect scheduling information with `perf-sched` for Linux. LDB timestamps the events with the same clock source as the external tool and stitches the events using the timestamp when analyzing them.

3.3.4 Analysis Script

LDB provides an analysis script to generate per-function statistics for collected latency samples. Further, it provides another analysis script that constructs a timeline of specific requests with function names and line numbers. It can stitch together the events generated by application threads (i.e., request and synchronization events), the stack scanner, and the OS scheduler (i.e., scheduling events).

Constructing the timeline for a specific request, identified by its request

ID, requires stitching together all events that occur during the processing of that request. Such a timeline can have multiple components, requiring the script to make multiple passes over the data generated by the profiler. First, the script looks through the event log until it observes the REQ_START event with the request ID, indicating the arrival of that request. The script tracks all FUNCTION events generated by the thread processing that request after the REQ_START event. Upon reaching a MUTEX_WAIT event, if the thread experiences non-negligible wait time (e.g., longer than 1 μs between the MUTEX_WAIT and MUTEX_LOCK events), the script scans the event log backward to identify the mutex holder thread by searching for a MUTEX_LOCK event with the same mutex address. Once the thread holding the mutex is identified, the script logs all FUNCTION events produced by that thread until it releases the mutex. Then, the script continues logging FUNCTION events by the original thread processing the request until it finds a REQ_BLOCK, REQ_END, or REQ_END_ALL event. The output of the script is a log of all events impacting the processing time of the request, each event identified by (name, thread ID, start time, end time) tuple. Such information can be easily visualized as shown in Figure 3.9.

3.4 Implementation

We implemented a prototype of LDB for the x86 architecture and the Linux environment. Our implementation has three components: 1) an extension to LLVM [13], called LLVM-LDB, to annotate call frames, 2) a stack scanner library to poll the generation numbers and calculate latency values, and 3) an API and bindings to capture request and synchronization events

automatically. Our implementation integrates with the Linux performance monitoring subsystem (`perf-sched`) to track context switches [94]. We also developed scripts to parse and analyze the data recorded by our tool. The core LDB tool is ≈900 lines of C code, the scripts are ≈1,200 lines of Python code, and the changes made to LLVM are ≈250 lines of C++ code. In this section, we describe more implementation-specific details for LDB.

3.4.1 LLVM-LDB

Reserving TLS. We reserve the two 8B TLS variables (generation number and the RBP of the last call frame) at a fixed offset from the TLS base address (FS base) with LLVM `ModulePass`. We make sure that LDB TLS variables are inserted into the TBSS section after all the in-application TLS variables are inserted so that LDB TLS variables are located at a fixed offset from the FS base.

Call frame instrumentation. We modified the sequence of function prologue and epilogue through changes to the LLVM X86 backend. In the function prologue, we decrement the RSP by 16 to reserve the space in a call frame before the RBP is pushed into the stack. After the RBP is updated to the current RSP value, we fill up the reserved stack space and update the TLS variables. First, we increment the generation number in TLS and copy it into the reserved space. Second, we set up the canary in the reserved stack space. Finally, now that the call frame is ready to be scanned, the prologue code updates the RBP of the last call frame in TLS to the current RBP so that the stack scanner can start scanning the stack from a newly created call frame.

In the function epilogue, we revert the instrumented operations in the prologue. First, before tearing down the call frame in the function epilogue,

the compiler first updates the RBP of the last call frame in TLS to avoid the current call frame being scanned while it is being destroyed. It then copies the saved RBP in the current call frame—which holds the RBP of the parent call frame—into the TLS region. After the saved RBP is popped from the stack with a standard epilogue sequence, RSP is incremented by 16 to destroy the reserved space for LDB. In total, we add 9 instrumentation instructions (7 in the prologue and 2 in the epilogue) which add less than 1 ns to each function call.

Thread instrumentation. We instrument the main function to initialize LDB using LLVM `ModulePass`. The initialization allocates the shared memory and per-thread event buffer before registering the main thread into the shared memory with its thread ID, FS base address, and event buffer address. Then, LDB launches the stack scanner thread and the logger thread. To initialize a newly launched thread and clean up the state before it exits, we interpose `pthread_create()`. Before a newly created thread executes its original start routine, LDB allocates the per-thread event buffer and registers the thread into the shared memory. After the original thread starts routine returns, it frees the event buffer and deregisters from the shared memory so that the exited thread is no longer scanned by the stack scanner.

3.4.2 The LDB API and Parameters

Request tagging API. LDB provides a way to annotate the threads with the following C APIs:

```
void ldb_req_start(uint64_t req_id, void *queue=NULL);
void ldb_req_block(uint64_t req_id, void *queue);
void ldb_req_end(uint64_t req_id);
```

```
void ldb_req_end_all();
```

When a thread starts to handle a request, the thread can be annotated with the request ID using `ldb_req_start()`. Optionally, if a request is dequeued from a software queue, the queue address can be specified. Multiple threads can be annotated with the same request ID with parallel processing, and a single thread can be annotated with multiple request IDs for batch processing. When a thread needs to enqueue a partially executed request into the queue the thread can hand off the responsibility of the request with `ldb_req_block()`. It indicates that the current thread is not responsible for the request anymore, but the current thread or another thread will resume processing the request later. If a thread finishes processing a request, it can clear the annotated request ID with `ldb_req_end()`. Alternatively, when a thread needs to clear all the annotated request IDs to the current thread, it can use `ldb_req_end_all()`. We decided to allow the programmer to specify the request ID, so that it can be correlated at the RPC level in coordination with other tools.

3.5 LDB Use Cases

To demonstrate the broad utility of LDB, we illustrates four use scenarios: visualizing a timeline of a specific request, debugging tail latency, debugging throughput, and studying the latency of specific functions. We evaluate these use scenarios with two latency-sensitive applications and one throughput-oriented application:

1. Memcached is a multithreaded, latency-sensitive, in-memory key-value store. We debug two different workloads: SET and GET. The SET workload exposes mutex and memory-intensive code paths. Each SET request

can access a global lock, `slabs_lock`, multiple times to allocate and free the memory and a hash table bucket lock, `items_lock`, to update the hash table. Additionally, when a Memcached memory is saturated, it may need to acquire `lru_lock` to evict stale items. On the other hand, a GET request only needs to acquire the `items_lock` before fetching a value from the hash table.

We allocated 10GB of memory for Memcached and used 100 million keys, evenly distributing them across requests. The value lengths are uniformly distributed between 4B and 1024B. We use the default hash power, which automatically grows based on the number of key-value pairs inserted into the hash table.

2. Lucene is a multithreaded, latency-sensitive in-memory search engine library [108]. Lucene's processing time is much longer than Memcached, helping us demonstrate the value of LDB under a variety of conditions. We used a dataset of 403,619 COVID-19-related tweets. Each client generates single-term search queries based on the word distribution in the dataset. For each search request, Lucene first retrieves the list of document IDs from `Segment`s where the mapping between a word and the list of document IDs is stored. Once the list of all relevant document IDs is retrieved, it fetches the pre-computed score (relevance between the document and the search query) for each document and returns the top 100 documents with high scores. As all shared data structures are protected by a mutex, requiring multiple mutex operations to serve each request.

3. Qperf [111] is a performance measurement tool for Quicly, Fastly's implementation of the QUIC protocol [58]. Unlike the other two applications,

	Peak Throughput	**LOC Changed for Tagging**
Latency-sensitive workloads		
Memcached SET	1.1 Mreqs / s	3
Memcached GET	3.0 Mreqs / s	3
Lucene	5.4 Kreqs / s	2
Best-effort workload		
Qperf	5.63 Gbps	4

Figure 3.8: Workload characteristics: peak throughput at 100% load without any profiling and the LOC changed for request tagging. Peak throughput is measured with 8 workers for Memcached and Lucene, and a single core for Qperf.

it measures the highest possible throughput between a server and a client. To achieve the highest throughput, we modified the original Qperf implementation to busy-poll the packets. This application helps showcase the value of LDB when measuring the average per-packet latency for each function, helping identify functions that harm the average throughput of the application. We use Reno as the congestion control algorithm and enable generic segment offload (GSO).

To use LDB on the applications listed above, we compiled the applications with `LLVM-LDB`. In addition, we inserted tagging annotations at each code location where a thread starts to handle a new request (Memcached and Lucene) or a new packet (Qperf). These points were easy to identify, and only required 2–4 LOC changes. Specifics are reported in Figure 3.8, along with the peak throughput of each workload.

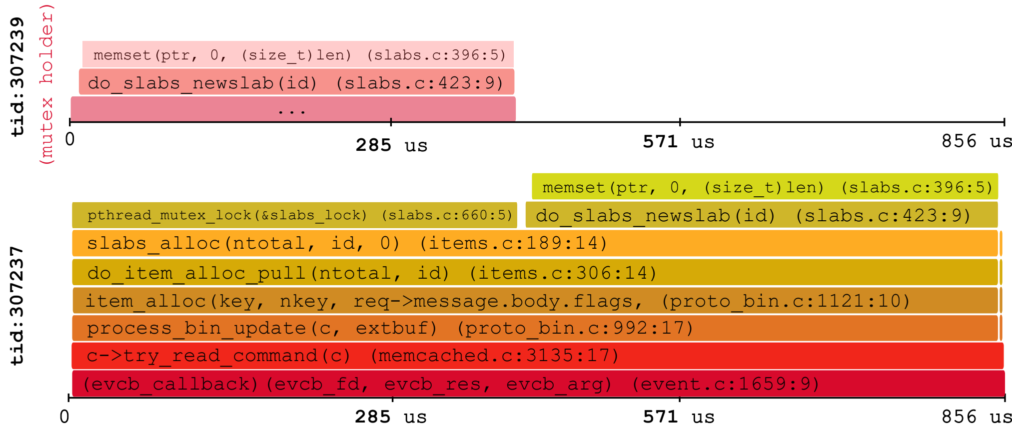


Figure 3.9: Timeline of the request of the longest request processing time in Memcached SET workload constructed by LDB.

3.5.1 Reconstructing the Timeline of the Request

When the application tags each request with a unique request ID, LDB can construct a timeline of any tagged request including interactions with other threads, which has been expensive with existing tools. Figure 3.9 shows an example of a timeline of a request SET workload in Memcached. We picked the request with the longest request processing time we observed during the initial slab allocation phase as the heap is populated because it provides a simple yet strong example of the value of LDB.

The detailed request timeline of LDB immediately shows what slowed down the processing of the request, including interactions with another request. When the request processing thread starts to handle the request, it waits for the `slabs_lock` mutex. LDB does not only tell the waiting time for the `slabs_lock` but also helps identify the thread holding the mutex and the function it's executing. In this example, the thread holding the mutex executes `memset()` while holding `slabs_lock`. After the thread processing

Function	p50	p99	p9999 ▽
slabs_alloc() ↳ pthread_mutex_lock(&slabs_lock)	< 1	13.57	22.00
do_item_unlink_nolock() ↳ STATS_LOCK()	< 1	6.93	20.51
lru_pull_tail() ↳ pthread_mutex_lock(&lru_locks[])	2.14	17.53	19.62
do_item_link() ↳ STATS_LOCK()	< 1	14.64	19.50
item_unlink_q() ↳ pthread_mutex_lock(&lru_locks[])	2.03	10.56	18.88

Figure 3.10: Latency statistics of top 5 functions (and its caller) ranked by 99.99th percentile latency in Memcached SET workload. All numbers are in μs.

the request acquires the lock, the dominant request processing time is spent executing `memset()` that took 645.7 μs. Such fine-grained tracing helps identify the main culprit which is performing `memset()` while holding the `slabs_lock`.

3.5.2 Tail Latency Debugging

With Memcached and Lucene, we demonstrate that LDB can list functions that contribute to high tail latency, giving an insight as to how to improve their tail latencies.

For Memcached SET workload, Figure 3.10 lists the top five tail-contributing functions of Memcached ranked by 99.99th percentile latency. All five functions perform locking. Three out of the five functions contend for global locks related to memory management (`slabs_lock`) and statistics collection (`stats_lock`). To fix the tail latency from the `slabs_lock`, one

Function	p50	p99	p9999 ▽
`resp_finish()` ↳ `THR_STATS_LOCK()`	< 1	9.04	18.55
`transmit()` ↳ `THR_STATS_LOCK()`	< 1	9.66	18.26
`do_item_get()` ↳ `assoc_find()`	< 1	10.10	18.25
`item_lock()` ↳ `mutex_lock(&item_locks[])`	< 1	10.12	17.19
`resp_start()` ↳ `memset()`	1.00	10.28	17.00

Figure 3.11: Latency statistics of top 5 functions (and its caller) ranked by 99.99th percentile latency in Memcached GET workload. All numbers are in μs.

could consider reducing contention by using a per-thread cache [5] and by zeroing memory without holding the lock. The `stats_lock`, on the other hand, could be fixed by either not using a lock, which would reduce accuracy, or by maintaining per-thread stats. Finally, the other two functions use the per-slab class lock, which is required for updating the LRU timestamp and evicting stale key-value pairs. To reduce the latency, one should fine-tune the chunk size growth factor (`-f`) based on the value length distribution.

For Memcached GET workload, Figure 3.11 shows that two of the top five functions are from per-worker thread locks (`THR_STATS_LOCK`). In Memcached, each network connection is assigned to one of the worker threads, but the requests can be processed by any worker thread. While the worker thread processes the request, it needs to acquire the lock of the worker thread that owns the network connection to update the statistics counters. When there is a small number of connections compared to the number of worker threads, or when the load is skewed to a subset of network connec-

Function	p50	p99	p9999 ▽
`IndexSearcher::search()` ↳ `Scorer::score()`	72.18	2,005	6,232
`Norm::bytes()` ↳ `IndexInput::readBytes()`	657.7	1,690	4,899
`boost::make_shared()` ↳ `new()`	30.59	61.60	78.61
`SegmentReader::docFreq()` ↳ `TermInfosReader::get()`	< 1	57.05	61.59
`TopDocsCollector::topDocs()` ↳ `populateResults()`	< 1	20.27	25.09

Figure 3.12: Latency statistics of top 5 functions (and its caller) ranked by 99.99th percentile latency in Lucene workload. All numbers are in μs.

tions, the per-worker thread statistics lock can be congested. The solutions mentioned above for `stats_lock` apply here too. Another two of the top five functions are for the hash table data structure (`assoc_find()` and per-bucket lock, `item_locks`). When a hash collision happens in the hash table, `assoc_find()` iterates over the bucket to find the item with the same key while holding the `item_lock`. One should consider initializing the Memcached with higher `hashpower`.

The last one is for memory operation to clear the allocated memory for a response. Considering that a response buffer will be overwritten with response data, one could consider removing the `memset()` operation, but care must be taken to avoid sending uninitialized data.

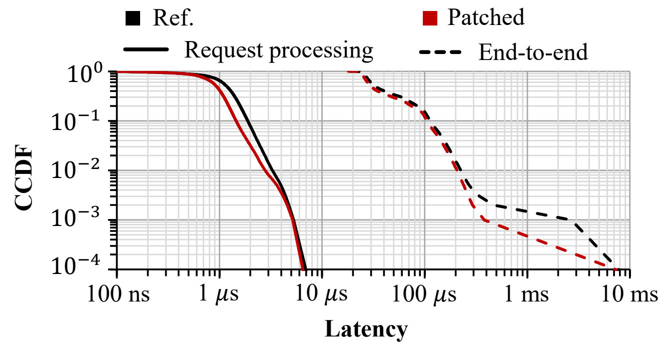
For Lucene workload, Figure 3.12 reports that the top two functions dominate the tail request processing time. Once Lucene receives a search query, it first fetches a list of document IDs by binary searching `Segments` after reading `Segments` in `IndexInput::readBytes()`. Once it has the list

of document IDs, it looks up the score (the relevance between the query and the document) for each document and enqueues the document ID with its score into the max heap tree in `Scorer::score()`. In this case, a tail latency problem arises because the most popular term in the dataset appears in 88,558 documents. Thus, `Scorer::score()` needs to iterate 88,558 times to look up the score and enqueue it into the max heap tree, which can take 6.2 ms. To reduce this latency, one could consider utilizing an increased level of parallelism [40]. That is, if the length of fetched document ID is too long, the search application could use multiple threads where each thread fetches the score of a subset of document IDs.

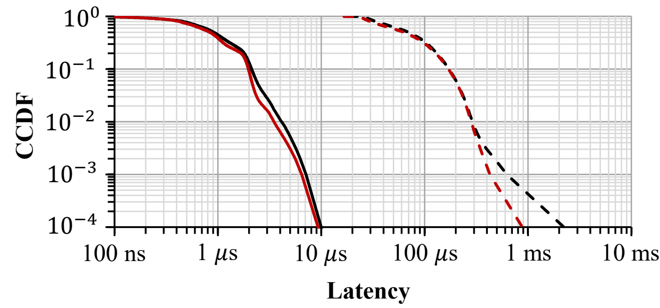
The other three functions are less significant. The memory allocation for reading the `Segment`s with `new()` takes up to 79 μs, fetching the score of a document with `get()` takes up to 62 μs, and popping the top 100 documents from the max heap tree in `populateResults()` takes up to 25 μs.

Actionable Insights. To demonstrate that LDB provides actionable insights that developers can use to improve the latency behavior of real applications, we patch Memcached and Lucene using the output of LDB. We show both the request processing time, revealing the improvement to just the part of the application that LDB can profile, and the end-to-end processing time, which includes other sources of tail latency like the kernel network stack and the network fabric.

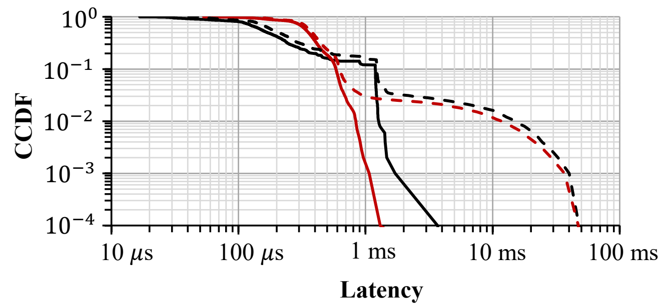
For Memcached, we (1) preallocate the slabs to avoid memory allocation while serving the request, (2) fine-tune the object size of each slab to avoid contention in slab classes by specifying minimum object size and adjusting chunk size growth factor, and (3) convert global and per-connection stats into per-thread stats. Figure 3.13 (a) and (b) show the request processing time and end-to-end latency distribution before/after applying the patch. Because multiple responses can be batched before written to the wire, the



(a) Memcached SET



(b) Memcached GET



(c) Lucene

Figure 3.13: Request processing latency and end-to-end latency after applying patches that fix latency problems identified by LDB.

improvement of end-to-end latencies is larger than the request processing times at some tail percentiles. The patch reduces the 99th percentile request processing time by 15% and 99th percentile end-to-end latency by 8% for SET workload; 99th percentile request processing time by 16% and 99th percentile end-to-end latency by 3% for GET workload.

For Lucene, we adjust the parallelism to serve the request. Specifically, we use four concurrent threads to serve each request. Figure 3.13 (c) shows the latency distribution of the request processing time and end-to-end latency before/after the patch. The increased parallelism can hurt performance for short requests due to the straggler effect, synchronization, and scheduling overhead, but results in large reduces to request latency in the tail. This problem has been studied extensively in prior work, which suggests an even more sophisticated fix would be to dynamically adjust parallelism based on the number of instantaneous requests in the system and the execution time [40]. The patch reduces 99th percentile request processing time by 34% and 99th percentile end-to-end latency by 13%.

3.5.3 Debugging Throughput of Qperf

We use LDB to debug the average performance of Qperf, demonstrating its value beyond tail-latency debugging. In particular, we profile the egress path on a Qperf server, focusing on the average per-packet latency, allowing us to determine an upper bound on achievable throughput. We find that each batch of 32 1500-byte packets takes 38.11 μs on average, putting a cap on throughput at around 9.8 Gbps. Note that actual throughput has to be lower because not all batches nor packets are maximum sized. Further, the server performs other functions beyond continuously transmitting data packets (e.g., process and transmit acknowledgments).

Function	Avg Latency ▽
`send_pending()` ↳ `send_dgrams()`	29.63 μs
`allocate_ack_eliciting_frame()` ↳ `do_allocate_frame()`	2.79 μs
`encrypt_packet()` ↳ `ptls_aead__do_encrypt()`	2.41 μs

Figure 3.14: Top 3 functions (and its caller) ranked by the highest average latency in Qperf workload. The average processing time for 32 packets is 38.11 μs.

We use LDB to identify which functions take the most time on average for transmission handling, revealing throughput bottlenecks. Figure 3.14 shows the top three functions with the highest average latency. The biggest bottleneck, responsible for 77.7% of the processing time of a batch, is `send_dgrams()` which transmits packets through the kernel's `sendmsg()`, showing that the biggest performance bottleneck lies in the kernel. Other bottlenecks include memory allocation in (`do_allocate_frame()`) and encryption (`encrypt_packet()`). The remaining processing time for a batch of packets can be attributed to a collection of lower-latency functions. Thus, to improve the throughput, one should optimize the network stack (e.g., by using kernel-bypass), memory operations, and cryptographic operations.

To highlight the value of the profile produced by LDB, we compare its output to the profile produced by Linux's `perf`. Figure 3.15 reports the list of function names ranked by highest CPU time by `perf`. It shows that `perf` cannot pinpoint any of `send_dgrams()`, `ptls_adad__do_encrypt()`, or `do_allocate` functions that are responsible for 91% of the packet processing time, reporting that they consume 0.04%, 0.28%, and 0.2% of the CPU time, respectively. In particular, `perf`'s focus on average CPU time provides

Function	CPU Time ▽
__libc_recvfrom	7.61%
send_pending	4.11%
quicly_send	2.56%
... (39 more functions) ...	
do_allocate_frame	0.28%
... (10 more functions) ...	
ptls_aead__do_encrypt	0.20%
... (152 more functions) ...	
send_dgrams	0.04%

Figure 3.15: Top 5 functions ranked by the highest CPU time in Qperf workload reported by Linux perf.

very coarse grain results, focusing on top-level functions like `quicly_send` which encapsulate all egress path functionality. Furthermore, it doesn't differentiate between functions on the critical path of egress traffic, and those happening periodically off the critical path, and it can't tie kernel delays to functions. Thus, we conclude that LDB can provide superior insights even when average performance is the focus of the debugging process.

3.6 Performance Evaluation

Our evaluations answer the following key questions:

1. How fine is the granularity of LDB's latency measurement?
2. Is LDB more portable than hardware-assisted latency debugging systems?
3. Can LDB limit the overhead it places on applications?
4. Can the trace data from LDB be decoded quickly?
5. How much does each component of LDB contribute to overhead?

Testbed. We use two machines with eighteen-core Intel Xeon Gold 6534 3.0GHz CPU (Ice Lake), 64GB RAM, and Mellanox ConnectX-6 200GbE NIC. For the portability experiment(§ 3.6.2), we compare its performance with Intel Broadwell machines (Intel Xeon E5 2640 v4 2.4GHz CPU, 64GB RAM, and Mellanox ConnectX-4 25GbE NIC) and AMD Zen3 Milan machines (AMD 7543 2.8GHz CPU, 256GB RAM, and Mellanox ConnectX-5 25GbE NIC). The median network RTT between two machines measured with ICMP packets is 30 μs. We use one machine as a server and the other as a client. Memcached and Lucene clients generate the requests following an open-loop Poisson arrival process, and Qperf clients generate a stream of requests for a data packet to measure the network bandwidth with TCP Reno as transport.

Applications. We use a synthetic application for microbenchmark. To evaluate the performance of LDB, we reuse the workloads used in §3.5; Memcached SET/GET and Lucene are latency-sensitive workloads, and Qperf is a throughput-oriented workload.

Baseline. We compare LDB to Intel Processor Trace (Intel PT) which backs state-of-the-art latency profilers [91, 106, 109], Coz that profiles the causal relationship between the function speedup and program speedup, and Xray that profiles the application's latency behavior with static timestamping. For Intel PT, we use `perf-intel-pt` provided by Linux to record and decode the Intel PT packets. For a fair comparison, we use a coarse-grained timing packet with `tsc` and decode only function call and return events with command line argument `--call-ret-trace`. We disable return compression (`noretcomp`) for more reliable decoding.

Evaluation Metrics. We report end-to-end latency (for latency-sensitive applications), average throughput (for best-effort application), raw trace size, and decoding time. End-to-end latencies and the average throughput

```
int worker(int recursion_level) {
  if (recursion_level == 1) {
    busy_loop(CALL_STACK_REFRESH_PERIOD);
  } else {
    worker(recursion_level - 1);
  }
}

int main() {
  while (true) {
    worker(CALL_STACK_DEPTH = 20);
  }
}
```

Figure 3.16: a synthetic application which returns and creates new CALL_STACK_DEPTH stack frames every CALL_STACK _REFRESH_ PERIOD.

are measured at the clients, and raw trace size and decoding time are measured at the server after the experiment finishes. Raw trace size measures the output size of each system, and decoding time measures the time required to parse the raw output to function-level latencies and to calculate the statistics of the function latencies. Because Intel PT takes too much time to decode, we measure the latency for decoding 1 ms long Intel PT trace. For LDB, we run the experiments for 4 seconds for Memcached and Qperf, and 1 minute for Lucene.

3.6.1 Microbenchmark

We delve into a detailed analysis of LDB's latency measurement granularity using a synthetic application described in Figure 3.16. This application repeatedly destroys and reconstructs 20 call stacks through recursion, with

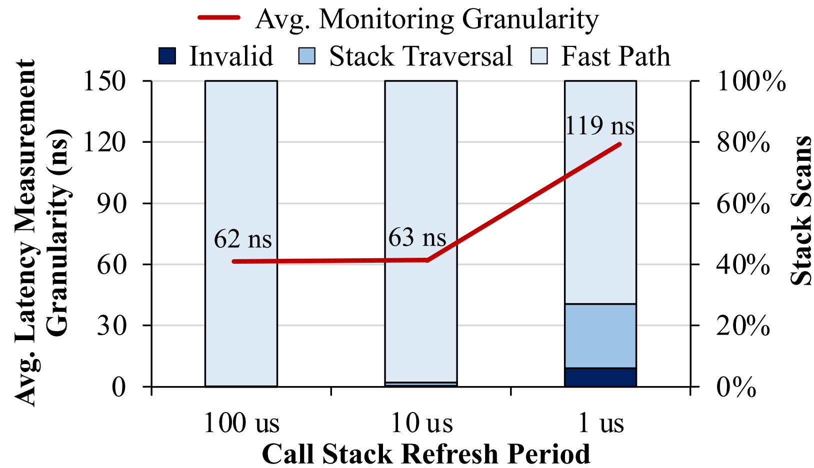


Figure 3.17: Average latency measurement granularity and the breakdown of stack scanning iterations with different call stack refresh periods in the synthetic application.

a predefined refresh period (CALL_STACK_REFRESH_PERIOD). We experiment with varying the call stack refresh period from 100 microseconds to 1 microsecond and measure the average latency measurement granularity, defined as the average time elapsed between two successive valid stack scans. We further categorize the stack scanning iterations into three groups: invalid scans resulting from sequential lock fails with data races, stack traversals, and fast path iterations where no modification is detected in either the most recent RBP or the generation number in the TLS region.

Figure 3.17 presents the results. As the call stack refresh period decreases, the application thread interacts with the stack frames more frequently to destroy existing call stacks with function returns and to build new ones with new function calls. This increased frequency leads to cache thrashing and data races between the application thread and the stack scanning thread

more often, increasing the average granularity in latency measurement with more invalid stack scans. In addition, with more frequent modifications in the stack frames, the stack scanner requires more iterations of full stack frame traversals, which further increases the average latency measurement granularity. In an experiment with a function depth of 20 and 1 μs call stack refresh period, the latency can be measured with the granularity of 119 ns with 6% invalid stack scans, 21% of full scans, and 73% of fast path scans.

When dealing with multiple application threads, the average granularity increases in proportion to the number of application threads being profiled. To attain more refined granularity in latency measurement, multiple stack scanner threads can be used, each profiling a subset of the application threads.

3.6.2 Portability of LDB

LDB is not designed for a specific platform. In principle, its design can be used on most architectures such as x86, ARM, and RISC-V. However, Intel-PT-based tools, such as NSight, are tied to Intel's specific architectures and cannot be ported to other platforms. Our LDB prototype is implemented for x86 architectures and works well on any x86 architectures while Intel PT only works with some Intel processors (later than Broadwell).

To illustrate the portability of LDB, we run the Qperf workload with different x86 CPU models and compare it against the reference (i.e., no latency profiling) and Intel PT. Figure 3.18 shows the average throughput measured by Qperf on three different CPU architectures. It shows that Intel PT's performance highly depends on the CPU architectures. Even though Intel PT has only a 4% of throughput drop on the recent Ice Lake Intel CPU, it experiences 59% of the throughput drop on an Intel Broadwell

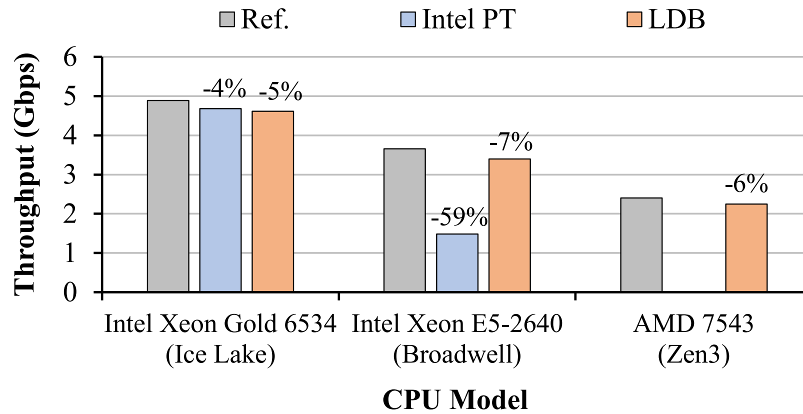


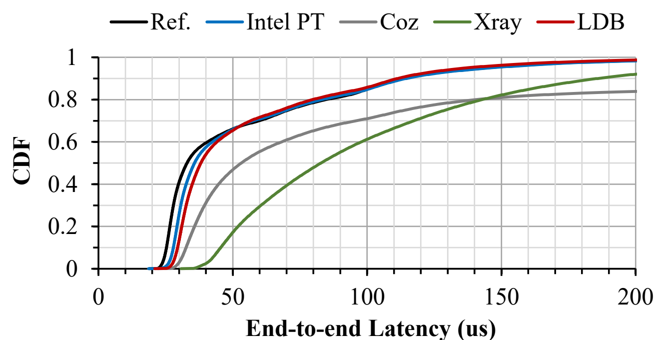
Figure 3.18: Average throughput of reference (without any profiling), Intel PT, and LDB with Qperf workload with different CPU architectures.

CPU, and it cannot be used for AMD processors. On the other hand, LDB has a more consistent overhead of up to 7% thanks to its software-based approach.

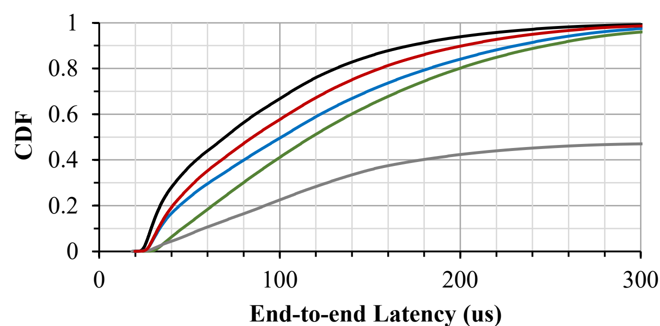
3.6.3 Overheads of LDB

Application performance degradation. To get more confidence in LDB's low overhead, we measure the application performance impact on three latency-sensitive workloads (Memcached SET/GET and Lucene) and compare it to other profiling mechanisms. For the benefit of Intel PT, benchmarks ran on our testbed with Intel Xeon Gold 6534.

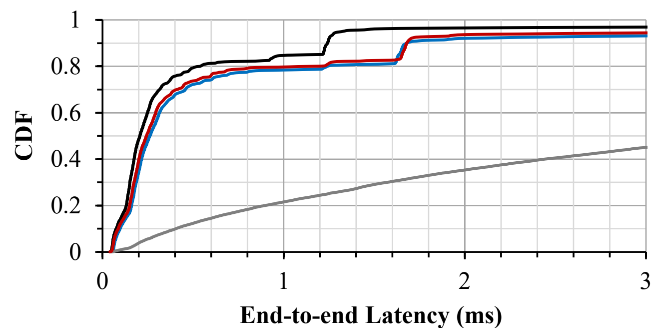
Figure 3.19 shows the end-to-end latency distribution measured at the client when the load is 20% of the system's capacity for Memcached and Lucene. We compare the performance of the applications when no latency profiling is done (i.e., Ref.) to when LDB, Intel PT, Coz, and XRay are



(a) Memcached SET



(b) Memcached GET



(c) Lucene

Figure 3.19: End-to-end latency distribution of reference (without any profiling), Intel PT, and LDB with Memcached SET, GET, and Lucene workload at 20% of load.

Workload		**Trace Size / s (trace errors / s)**	**Decoding Time / s**
Memcached SET	Intel PT	696.84 MB (2k trace errors)	48.4 min
	LDB	149.38 MB (-79%)	1.7 s
Memcached GET	Intel PT	796.72 MB (5k trace errors)	1.8 hr
	LDB	237.46 MB (-70%)	2.7 s
Lucene	Intel PT	1,066.29 MB (6k trace errors)	3.1 min
	LDB	2.12 MB (-99%)	0.7 s
Qperf	Intel PT	944.03 MB (559k trace errors)	3.7 hr
	LDB	25.4 MB (-97%)	0.8 s

Figure 3.20: Trace size and decoding time of Intel PT and LDB for four workloads. Trace size and decoding time are normalized by execution time.

used. For all workloads, Coz has the largest overhead at tail because it intentionally delays all the other threads than the thread being sampled, which makes it impractical to use over live traffic. The overhead of XRay is proportional to the number of function invocations as it statically instruments every function entry/exit to measure the latency. Due to its high overhead, the load exceeds the capacity, leading to extremely high latency with high queueing delay. Intel PT and LDB have comparable overhead across the workloads. LDB increases median(99th percentile) latency by 16%(1%), 22%(10%), and 18%(43%) for Memcached SET, GET, and Lucene workloads while Intel PT increases 9%(2%), 45%(23%), and 27%(64%) in the same setting.

Trace size and decoding time. Figure 3.20 reports the output trace size and decoding time of Intel PT and LDB for the three applications. Intel

PT requires high memory / PCIe bandwidth and disk space, especially for applications with more branches and jump instructions. For example, in the case of Qperf, Intel PT outputs 944 MB/s of trace data. In addition, because of the limited memory bandwidth, it drops the event records and results in up to 559 thousand trace errors per second which make its visibility limited. To make matters worse, Intel PT takes up to 3.7 hours to decode 1 second of trace data, converting raw branch and jump information into function-level latencies. In contrast, the size of LDB trace is up to 99% smaller than Intel PT, typically requiring less than 250 MB/s, and it only takes a few seconds to decode 1 second of trace data.

3.6.4 Breakdown of LDB's overhead

We analyze how much each component contributes to the overhead for Lucene workload with the highest latency distortion under LDB whose median(99th percentile) latency is increased by 35%(69%). We gradually activate four components of LDB: application instrumentation (inst), the stack scanning/logging thread (scan), the shim layer (shim), and Linux scheduling event recording.

Figure 3.21 shows that instrumentation, the stack scanning, the shim layer, and Linux scheduling event recording are responsible for 11% (23%), 7% (17%), 3% (14%), and 1% (0%) of the median (99th percentile) latency increase, respectively.

3.7 Related Work

Sampling-based tools. Today's tools based on statistical sampling like perf are unsuitable for studying tail latency because they gather samples

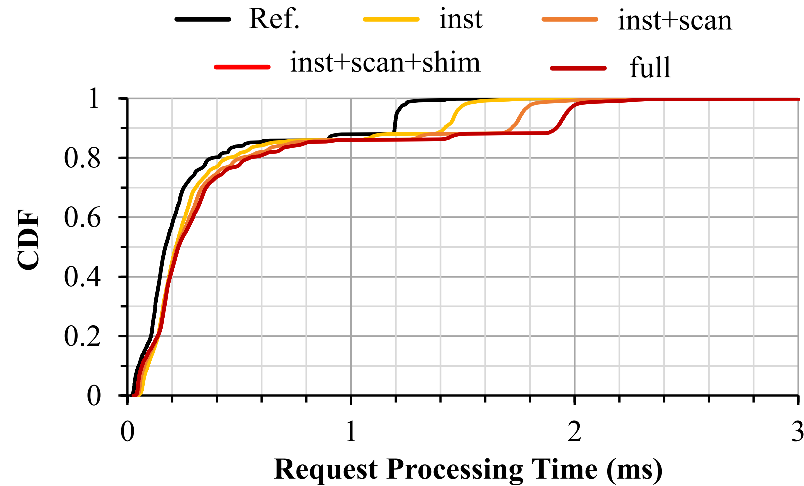


Figure 3.21: Performance Breakdown of LDB with Lucene workload at 20% load.

too infrequently, and focus on average or steady-state performance. Coz aims at finding the bottleneck function whose improvement can translate to the most application speedup. Coz statistically estimates the speed of the application (unlike perf estimating CPU time) and performs dynamic experiments to capture the effect of thread interplay. Coz conducts several short online experiments, adding delays to all-but-one threads to simulate the virtual speedup of one function. Although Coz addresses some limitations of perf, it is still inappropriate for tail latency debugging. First, Coz cannot observe tail behavior because it relies on statistical sampling for estimating average speedup. Second, for the lines of code Coz can identify, it cannot provide a deeper explanation of why it is a bottleneck.

Trace-based latency profiling tools. Another strategy is to use a trace-based tool that can capture the time spent in every function. However, these approaches either cause large slowdowns to the application (Xray), or

generate too much trace data, prohibiting interactive analysis (Intel PT). For example, XRay [47] is a trace-based tool that records the execution time of individual functions through compile-time instrumentation. XRay's instrumentation provides a rich timeline of the execution of functions. It's important to note that going from those raw traces to latency-related information is nontrivial. Further, XRay's instrumentation introduces considerable overhead to the application, limiting its applicability to tracing in production.

Intel PT, on the other hand, is a hardware offload that can reduce profiling overhead while observing an entire program at once. It is capable of recording every control flow operation (calls, branches, jumps, etc.) to an in-memory log. Although Intel PT has many potential use cases (e.g., reverse debugging [63]), NSight was one of the first systems to use Intel PT for debugging tail latency behaviors [91]. NSight primarily focuses on studying the host networking stack, but it includes a hard-coded understanding of requests for specific workloads like Memcached. Despite its advantages over existing approaches, NSight still has limited visibility. For example, it cannot observe cross-thread interactions, and it requires significant per-application tuning, limiting its applicability. MagicTrace (another tool built on Intel PT) focuses more on debugging very short duration time segments of application code [109]. Both systems can generate precise timelines for program execution.

Dependence on Intel PT imposes significant limitations in terms of the platforms they can support, the rate of data generated, and the time to decode it. For example, Intel PT can generate up to one GB/s of samples, requiring RAM or high-speed flash to record it. Moreover, it can take up to a few hours to decode a single second worth of samples. These limitations make tools based on Intel PT unusable for long-running applications or

interactive debugging. Finally, in terms of portability, Intel PT's overhead varies across generations of Intel CPUs. For example, our experiments show that the slowdown caused by Intel PT is considerable on older generations of Intel CPUs (§ 3.6.2).

Sampling with a busy-polling thread. SHIM collects hardware performance counters and software tags with busy running threads [45]. It shares LDB's basic strategy of sampling with busy-running cores, but it lacks the ability to measure invocation latency without additional mechanisms, such as our proposed stack sampling techniques. Moreover, a single busy-running SHIM profiler thread is needed for each hyperthread pair, resulting in high overheads due to competition over shared functional units. LDB, by contrast, can avoid this overhead by using only one monitor thread to profile multiple threads running across multiple cores.

Limiting tracing to specific functions. As seen with the case of Xray, timestamp instrumentation at each function's entry and exit entails significant overhead, especially for applications with many function invocations. Thus, limiting tracing to a few specific functions at a time could be necessary. There are various techniques and tools to enable dynamically enable/disable timestamp instrumentation: notably, dynamic instruction patching [11, 20, 47], dynamic instrumentation via eBPF [51, 89, 105], and instrumentation via JIT compiler [93]. However, because the scope of functions being profiled is limited, they require multiple iterations with the developer's hands-on interaction to pinpoint which functions are responsible for high latency, and they sacrifice the ability to capture complete timelines. There are efforts to streamline these iterations [21, 56, 113]. AMD offers a suite of profiling tools (e.g., Omnitrace [102] and uProf [97]). Both solutions rely on sampling. Further, Omnitrace offers Coz-like functionality as well as specific function instrumentation. Omnitrace's instrumentation adds 1024

instructions per function compared to LDB's 9 instructions per function.
Distributed latency tracing. Envoy tracing [81], Zipkin [96], Jaeger [92], AWS X-Ray [88], and Apache SkyWalking [87] provide a tool to trace a request in distributed computing environment at an RPC or microservice-level granularity. Distributed latency tracing tools may find which service is causing high end-to-end latency, but they don't have visibility inside the service. Distributed tracing systems and LDB are complimentary. Problematic services can be found with distributed tracing, while problematic functions in a specific service can be spotted with LDB.
Mutex bindings. Dynamic data race detectors, like Eraser [3], often use similar mechanisms to interpose on locking functions, but their goal is to instead verify if the application follows a consistent locking protocol.

3.8 Conclusion

In this chapter, we presented LDB, an efficient latency profiling tool with low overhead, high visibility, fast decoding speed, and good portability. It utilizes a key technique, stack sampling, where each function's invocation latency is measured by sampling a unique generation number assigned in the call frame. With optional request tagging by the developer, LDB can construct detailed timelines of each request, including cross-thread interactions caused by synchronization, the time spent in functions, and the contribution of the OS scheduler. Our evaluation demonstrates that LDB can profile the latency behavior of three applications and reveal their main performance bottlenecks effectively over multiple platforms with low overhead.

4 Future Work

Breakwater and LDB make substantial progress in mitigating and diagnosing compute congestion, facilitating microsecond-scale datacenter RPCs that adhere to strict SLO requirements. Nevertheless, this is merely one chapter in a broader narrative. The potential exists to further refine and expand Breakwater and LDB by exploring additional factors that were beyond the scope of this thesis.

4.1 Overload Control

4.1.1 Overload control for multi-layer services

In this thesis, we only consider a single-layer, single-server overload control scenario. Breakwater's receiver-driven, credit-based approach can be applied to multiple layers of microservices, preventing overload at each individual layer. However, when overload occurs in an intermediate layer of a multi-layer service, the work performed in earlier layers is wasted. We leave propagating overload signals and coordinating overload control across several layers of microservices for future work.

4.1.2 Delay from other types of congestion

Breakwater's primary emphasis is on addressing CPU congestion and lock contention, with a comprehensive evaluation of these specific scenarios. Nonetheless, a system may encounter bottlenecks from other resources, including memory bandwidth, PCIe bandwidth, and disk I/O. Some of these congestions might be manageable through the admission control with a performance-driven efficiency overload signal, but further investigation is needed to ensure both low latency and high utilization when these resources become a bottleneck.

4.2 Tail Latency Profiling Tool

4.2.1 Distributed latency profiling

Currently, LDB focuses on profiling the latency behavior of an application within a single server. However, we believe that the potential for LDB can extend beyond this scope. By integrating LDB's principle with existing distributed latency tracing tools, it could be developed to encompass a more comprehensive analysis of a request's latency behavior, including inter-node interactions and communications. Such an expansion would provide deeper insights into a request's lifetime over the entire distributed system.

4.2.2 Detailed analysis of a function with high tail latency

LDB can identify the specific function calls responsible for high tail latency, yet it falls short in explaining the underlying reasons *why* a particular func-

tion call takes a long time to execute, apart from delays caused by context switches. By integrating LDB with additional hardware and software counters, we believe it can be enhanced to provide more granular insights into the factors contributing to extended execution times for a function call. Such information could include interrupts, false sharing, thermal throttling, cache misses, page faults, and other underlying factors.

5 Conclusions

Due to the advancements in datacenter networks and the end of Dennard scaling, compute resources have become the primary factor determining the latency of datacenter applications. As a result, the optimization of latency performance through the mitigation of compute congestion has emerged as an essential task to satisfy the increasingly stringent SLOs for end-to-end latency. This dissertation identified two specific opportunities to reduce tail latencies of microsecond-scale RPC under compute congestion, with a particular emphasis on CPUs and locks.

Breakwater offers a robust solution to mitigate tail latency during server overload, whether triggered by CPU congestion or lock contention. It effectively detects the server overload by request queueing delay overload signal for CPU-related congestion and performance-driven efficiency overload signal for scenarios involving lock contention. In parallel, LDB furnishes insightful statistics concerning the application's tail latency behaviors and delivers a comprehensive time-series analysis of an individual RPC request. This functionality empowers developers to precisely pinpoint the functions contributing to high tail latency with minimal overhead.

Bibliography

[1] Chandramohan A Thekkath et al. "Implementing network protocols at user level". In: *SIGCOMM*. 1993.

[2] Jeffrey C Mogul and KK Ramakrishnan. "Eliminating receive livelock in an interrupt-driven kernel". In: *ACM Transactions on Computer Systems* (1997).

[3] Stefan Savage et al. "Eraser: A dynamic data race detector for multithreaded programs". In: *TOCS* (1997).

[4] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. "IO-Lite: a unified I/O buffering and caching system". In: *TOCS* (2000).

[5] Jeff Bonwick and Jonathan Adams. "Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources". In: *Proceedings of the General Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA*. USENIX, 2001, pages 15–33.

[6] Huamin Chen and Prasant Mohapatra. "Session-based overload control in qos-aware web servers". In: *INFOCOM*. 2002.

[7] Ludmila Cherkasova and Peter Phaal. "Session-based admission control: A mechanism for peak load management of commercial web sites". In: *IEEE Transactions on Computers* 51.6 (2002), pages 669–685.

[8] Matt Welsh and David Culler. "Overload management as a fundamental service design primitive". In: *SIGOPS European Workshop*. 2002.

[9] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. "Web Search for a Planet: The Google Cluster Architecture". In: *IEEE Micro* 23.2 (2003), pages 22–28.

[10] Matt Welsh and David E Culler. "Adaptive Overload Control for Busy Internet Servers." In: *USENIX Symposium on Internet Technologies and Systems*. 2003.

[11] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. "Dynamic Instrumentation of Production Systems". In: *ATC*. 2004.

[12] Sameh Elnikety et al. "A method for transparent admission control and request scheduling in e-commerce web sites". In: *International conference on World Wide Web*. 2004.

[13] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *CGO*. 2004.

[14] Luiz André Barroso and Urs Hölzle. "The case for energy-proportional computing". In: *Computer* 40.12 (2007), pages 33–37.

[15] Will Reese. "Nginx: the high-performance web server and reverse proxy". In: *Linux Journal* (2008).

[16] *More on today's Gmail issue*. `https://gmail.googleblog.com/2009/09/more-on-todays-gmail-issue.html`. 2009.

[17] Mohammad Alizadeh et al. "Data Center TCP (DCTCP)". In: *SIGCOMM*. 2010.

[18] Theophilus Benson, Aditya Akella, and David A. Maltz. "Network Traffic Characteristics of Data Centers in the Wild". In: *IMC*. 2010.

[19] Ming Mao, Jie Li, and Marty Humphrey. "Cloud auto-scaling with deadline and budget constraints". In: *GridCom*. 2010.

[20] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall Professional, 2011.

[21] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. "Catch me if you can: performance bug detection in the wild". In: *OOPSLA*. 2011.

[22] Berk Atikoglu et al. “Workload analysis of a large-scale key-value store”. In: *SIGMETRICS*. 2012.

[23] Rishi Kapoor et al. “Chronos: Predictable low latency for data center applications”. In: *SoCC*. 2012.

[24] Kathleen Nichols and Van Jacobson. “Controlling queue delay”. In: *Communications of the ACM* (2012).

[25] Irene Aldridge. *High-frequency trading: a practical guide to algorithmic strategies and trading systems*. Volume 604. John Wiley & Sons, 2013.

[26] Jeremy Cloud. “Decomposing twitter: Adventures in service-oriented architecture”. In: *QCon New York*. 2013.

[27] Jeffrey Dean and Luiz André Barroso. “The tail at scale”. In: *Communications of the ACM* (2013).

[28] Rajesh Nishtala et al. “Scaling memcache at facebook”. In: *NSDI*. 2013.

[29] Stephen Tu et al. “Speedy Transactions in Multicore In-Memory Databases”. In: *SOSP*. 2013.

[30] Yuehai Xu et al. “Characterizing facebook’s memcached workload”. In: *IEEE Internet Computing* (2013).

[31] Adam Belay et al. “IX: a protected dataplane operating system for high throughput and low latency”. In: *OSDI*. 2014.

[32] Aleksandar Dragojević et al. “FaRM: Fast remote memory”. In: *NSDI*. 2014.

[33] Anshul Gandhi et al. “Adaptive, model-driven autoscaling for cloud applications”. In: *ICAC*. 2014.

[34] EunYoung Jeong et al. “mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems”. In: *NSDI*. 2014.

[35] Anuj Kalia, Michael Kaminsky, and David G. Andersen. “Using RDMA Efficiently for Key-Value Services”. In: *SIGCOMM*. 2014.

[36] Ilias Marinos, Robert NM Watson, and Mark Handley. "Network stack specialization for performance". In: *SIGCOMM* (2014).

[37] Dmitry Namiot and Manfred Sneps-Sneppe. "On micro-services architecture". In: *International Journal of Open Information Technologies* (2014).

[38] Infiniband Trade Association. *InfiniBandTM Architecture Specification Volume 1 Release 1.3*. 2015.

[39] Mo Dong et al. "PCC: Re-architecting congestion control for consistent high performance". In: *NSDI*. 2015.

[40] Md. E. Haque et al. "Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services". In: *ASPLOS*. 2015.

[41] David Lo et al. "Heracles: Improving resource efficiency at scale". In: *ISCA*. 2015.

[42] Ben Maurer. "Fail at scale". In: *Queue* (2015).

[43] John Ousterhout et al. "The RAMCloud storage system". In: *ACM Transactions on Computer Systems* (2015).

[44] Simon Peter et al. "Arrakis: The operating system is the control plane". In: *TOCS* (2015).

[45] Xi Yang, Stephen M Blackburn, and Kathryn S McKinley. "Computer performance microscopy with shim". In: *ISCA* (2015).

[46] Yibo Zhu et al. "Congestion control for large-scale RDMA deployments". In: *SIGCOMM*. 2015.

[47] Dean Michael Berris et al. *XRay: A Function Call Tracing System*. Technical report. 2016.

[48] Betsy Beyer et al. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 2016.

[49] Neal Cardwell et al. "BBR: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time". In: *Queue* (2016).

[50] *Doorman: Global Distributed Client Side Rate Limiting.* `https://github.com/youtube/doorman`. 2016.

[51] Brendan Gregg. *Linux BPF Superpowers.* Mar. 5, 2016.

[52] Chuanxiong Guo et al. "RDMA over commodity ethernet at scale". In: *SIGCOMM*. 2016.

[53] Han Cai et al. "Real-time bidding by reinforcement learning in display advertising". In: *WSDM*. 2017.

[54] Inho Cho, Keon Jang, and Dongsu Han. "Credit-scheduled delay-bounded congestion control for datacenters". In: *SIGCOMM*. 2017.

[55] Mark Handley et al. "Re-architecting datacenter networks and stacks for low latency and high performance". In: *SIGCOMM*. 2017.

[56] Jiamin Huang, Barzan Mozafari, and Thomas F Wenisch. "Statistical analysis of latency through semantic profiling". In: *EuroSys*. 2017.

[57] Akamai Inc. *The State of Online Retail Performance.* `https://s3.amazonaws.com/sofist-marketing/State+of+Online+Retail+Performance+Spring+2017+-+Akamai+and+SOASTA+2017.pdf`. 2017.

[58] Adam Langley et al. "The quic transport protocol: Design and internet-scale deployment". In: *SIGCOMM*. 2017.

[59] Greg Linden. *Marissa Mayer at Web 2.0.* `http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html`. 2017.

[60] George Prekas, Marios Kogias, and Edouard Bugnion. "Zygos: Achieving low tail latency for microsecond-scale networked tasks". In: *SOSP*. 2017.

[61] Lalith Suresh et al. "Distributed resource management across process boundaries". In: *SoCC*. 2017.

[62] Tao Zhang et al. "Tuning the aggressive TCP behavior for highly concurrent HTTP connections in intra-datacenter". In: *Transactions on Networking* (2017).

[63] Weidong Cui et al. "REPT: Reverse Debugging of Failures in Deployed Software". In: *OSDI*. 2018.

[64] Kim M. Hazelwood et al. "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective". In: *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. IEEE Computer Society, 2018, pages 620–629.

[65] Marios Kogias and Edouard Bugnion. "Flow control for latency-critical rpcs". In: *KBNets*. 2018.

[66] Behnam Montazeri et al. "Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities". In: *SIGCOMM*. 2018.

[67] Henry Qin et al. "Arachne: core-aware thread management". In: *OSDI*. 2018.

[68] Akshitha Sriraman and Thomas F. Wenisch. "μTune: Auto-Tuned Threading for OLDI Microservices". In: *OSDI*. 2018.

[69] Stephen Yang, Seo Jin Park, and John Ousterhout. "NanoLog: A Nanosecond Scale Logging System". In: *ATC*. 2018.

[70] Hao Zhou et al. "Overload control for scaling wechat microservices". In: *SoCC*. 2018.

[71] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. "RPCValet: NI-Driven Tail-Aware Balancing of μs-Scale RPCs". In: *ASPLOS*. 2019.

[72] Dmitry Duplyakin et al. "The design and operation of CloudLab". In: *ATC*. 2019.

[73] Yoav Einav. *Amazon Found Every 100ms of Latency Cost them 1% in Sales*. `https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales`. 2019.

[74] Marios Kogias et al. "R2P2: Making RPCs first-class datacenter citizens". In: *ATC*. 2019.

[75] Bradley C Kuszmaul et al. "Everyone Loves File: File Storage Service (FSS) in Oracle Cloud Infrastructure". In: *ATC*. 2019.

[76] Amy Ousterhout et al. "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads". In: *NSDI*. 2019.

[77] Matteo Aquilina, Eric B Budish, and Peter O'Neill. *Quantifying the high-frequency trading" arms race": A simple new methodology and estimates*. Technical report. EconStor, Working Paper, 2020.

[78] *AWS Auto Scaling*. `https://aws.amazon.com/autoscaling/`. 2020.

[79] Juan M. Banda et al. *A large-scale COVID-19 Twitter chatter dataset for open scientific research - an international collaboration*. Version 91. `https://doi.org/10.5281/zenodo.3723939`. May 2020.

[80] Inho Cho et al. "Overload Control for μs-scale RPCs with Breakwater". In: *OSDI*. 2020.

[81] *Envoy Proxy*. `https://www.envoyproxy.io/`. 2020.

[82] *HAProxy*. `http://www.haproxy.org/`. 2020.

[83] Marios Kogias and Edouard Bugnion. "Tail-tolerance as a Systems Principle not a Metric". In: *APNet*. 2020.

[84] Gautam Kumar et al. "Swift: Delay is simple and effective for congestion control in the datacenter". In: *SIGCOMM*. 2020.

[85] YoungGyoun Moon et al. "AccelTCP: Accelerating Network Applications with Stateful TCP Offloading". In: *NSDI*. 2020.

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

[87] *Apache SkyWalking*. `https://skywalking.apache.org/`. 2022.

[88] *AWS X-Ray*. `https://aws.amazon.com/xray/`. 2022.

[89] *bpftrace: High-level tracing language for Linux systems*. `https://bpftrace.org/`. 2022.

[90] *Challenges in building a scalable Demand Side Platform (DSP) service*. `https://www.moloco.com/r-d-blog/challenges-in-building-a-scalable-demand-side-platform-dsp-service`. 2022.

[91] Roni Haecki et al. "How to diagnose nanosecond network latencies in rich end-host stacks". In: *NSDI*. 2022.

[92] *Jaeger: open source, end-to-end distributed tracing*. https://www.jaegertracing.io/. 2022.

[93] Yu Luo et al. "Hubble: Performance Debugging with In-Production, Just-In-Time Method Tracing on Android". In: *OSDI*. 2022.

[94] *perf: Linux profiling with performance counters*. https://perf.wiki.kernel.org/index.php/Main_Page. 2022.

[95] Timothy Stamler et al. "zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO". In: *OSDI*. 2022.

[96] *Zipkin*. https://zipkin.io/. 2022.

[97] *AMD uProf*. https://www.amd.com/en/developer/uprof.html. 2023.

[98] *ChatGPT*. https://chat.openai.com. 2023.

[99] Inho Cho et al. "Protego: Overload Control for Applications with Unpredictable Lock Contention". In: *NSDI*. 2023.

[100] *Google Bard*. https://bard.google.com. 2023.

[101] NVidia. *ConnectX-7 400G Adapters*. https://nvdam.widen.net/s/csf8rmnqwl/infiniband-ethernet-datasheet-connectx-7-ds-nv-us-2544471. 2023.

[102] *Omnitrace: Application Profiling, Tracing, and Analysis*. https://github.com/AMDResearch/omnitrace/. 2023.

[103] Adrian Cockroft. *Microservices Workshop: Why, what, and how to get there*. http://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference.

[104] IEEE DCB. *802.1Qbb - Priority-based Flow Control*. http://www.ieee802.org/1/pages/802.1bb.html.

[105] *eBPF*. https://ebpf.io/.

[106] *Fix performance bottlenecks with Intel VTune Profiler.* https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[107] *High-Performance, Feature-Rich NetXtreme® E-Series Dual-Port 100G PCIe Ethernet NIC.* https://www.broadcom.com/products/ethernet-connectivity/network-adapters/100gb-nic-ocp/p2100g.

[108] Ben van Klinken. *Lucene++.* https://github.com/luceneplusplus/LucenePlusPlus.

[109] *magic-trace: Diagnosing tricky performance issues easily with Intel Processor Trace.* https://blog.janestreet.com/magic-trace/.

[110] *Memcached.* http://memcached.org/.

[111] *qperf: performance measurement tool for QUIC.* https://github.com/rbruenig/qperf/.

[112] *Sequence counters and sequential locks.* https://docs.kernel.org/locking/seqlock.html.

[113] *wachy: A new approach to performance debugging.* https://rubrikinc.github.io/wachy/.