Layered Contention Mitigation for Cloud Storage

Meng Wang, Cesar A. Stuardo, Daniar Heri Kurniawan, Ray A. O. Sinurat, and Haryadi S. Gunawi
Department of Computer Science
University of Chicago
Chicago, USA
{wangm12, castuardo, daniar, rayandrew, haryadi}@uchicago.edu

***Abstract*—We introduce an ecosystem of contention mitigation supports within the operating system, runtime and library layers. This ecosystem provides an end-to-end request abstraction that enables a uniform type of contention mitigation capabilities, namely request cancellation and delay prediction, that can be stackable together across multiple resource layers. Our evaluation shows that in our ecosystem, multi-resource storage applications are faster by 5-70% starting at 90P (the 90^{th}percentile) compared to popular practices such as speculative execution and is only 3% slower on average compared to a best-case (no contention) scenario.**

***Index Terms*—Distributed Systems; Resource Management; Performance and Reliability; Fault Tolerance; Cloud Computing**

I. Introduction

Today, many varieties of products are advertised not only with traditional metrics such as throughput and average latency, but tail latency as well (e.g. Xms latency guaranteed at the Y^{th} percentile). In cloud deployments, resource sharing is a de-facto configuration and contributes as a dominant factor of unpredictable latency. Contention appears in many different resource layers, all directly impacting software systems that use multiple resources, including storage systems.

Let's take distributed cloud storage such as key-value stores as an example. They require CPUs to process user requests, memory to cache the data, lock resources to implement concurrent data sharing correctly, and storage devices to fairly and promptly serve their I/Os. However, CPUs might not be instantly available due to process/VM contention, load imbalance or task rebalancing across cores [1–8]; memory access can be halted by the language runtime for heap reorganization [9–14]; foreground locks might be used by background management operations such as compaction, flushing and migration [15–18]; and I/Os can be delayed under bursty workloads [19–25]. Another challenge on top of all of these is that unlike compute jobs that run for seconds, the small latency tail that storage users expect is at the millisecond granularity (e.g. 5ms at the 99^{th}-percentile latency).

Guaranteeing highly stable latencies in multi-resource systems including distributed storage is still an open-ended challenge. In this context, we studied and reviewed popular contention mitigation scenarios, from application modification, speculation, replica selection and resource-level optimization, and evaluate them on multiple dimensions such as simplicity, efficiency, reactivity and coverage. We found that while each of these methods has advantages in multiple dimensions, they have inherent limitations that cannot be fixed within its own category (more details in Section II).

We pose the following question: Is there a strategy that can combine the best of all the worlds, e.g. a strategy that can keep the simplicity of speculation, the efficiency of application modification, the reactivity of resource optimization, and the coverage of replica selection? This question is fundamentally hard to answer when existing methods solve the problem either entirely in the applications or in the individual resource managers (e.g. in OS, runtime or library).

We introduce LIBROS, an ecosystem for contention mitigation with supports from library, runtime, and operating system layers. The principle behind our ecosystem is that while distributed applications are responsible for the retry mechanisms (as they know where data replicas are), resource managers should help notify applications when resources are highly contended. In this "app-OS" co-design, neither the application nor the resource managers attempt to solve the problem entirely by itself.

The key ingredient in LIBROS is enabling a uniform type of support that can be adopted across multiple resource layers. For this, LIBROS first introduces an end-to-end "`request`" abstraction that flows through multiple resource layers. The concept of `request`, including its corresponding deadline and cancellability, becomes a first-class citizen. In mitigating tail latency, resource managers now operate on `request` level as opposed to opaque stream of bytes. Around this abstraction, resource managers can build a stackable support, namely request cancellation and delay prediction, for the individual resources that they manage.

To adhere on simplicity, our approach does not modify resource-level policies such as scheduling and allocation decisions. Instead, given a particular QoS policy that a resource manager employs, we build a delay predictor that estimates how long an incoming request will be delayed in that layer. Hence, all the resource layers in the LIBROS ecosystem can provide a new, uniform capability: when a request arrives at a resource layer, the resource manager predicts the delay in that layer, and if the delay violates the request deadline, the layer will cancel this (cancellable) request, knowing that the

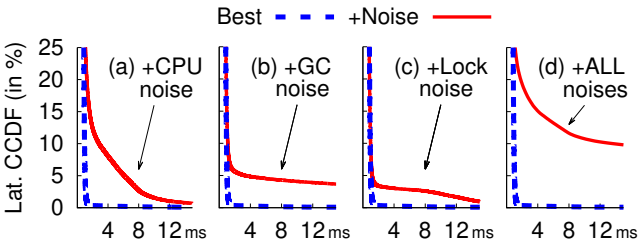

Fig. 1. **Multi-layer tails.** The figures, in CCDF format, show tail latencies due to (a) CPU contention, (b) GC pauses, (c) lock unavailability, and (d) all combined.

TABLE I
State-of-the-art (§II-B) vs. LIBROS §III-B) benefits.

	App-Simplicity	Efficiency	Reactivity	RL-Simplicity	Coverage
App. modification	—	√	√	√	—
Speculation	√	—	√	√	√
Replica selecion	√	√	—	√	√
Resource optimization	√	√	√	—	—
LIBROS	√	√	√	√	Major**

application has another replica to go to (§III).

To achieve fast reactivity, resource managers send cancellation notifications that inform the application to quickly react to the delay by sending speculative retries to other replicas. At the same time, efficiency (no extra load) is achieved because the original request has been automatically cancelled by the contended resource layer. Finally, for coverage, we build the capability above for three major often-contended resources. Specifically, we present ETOS, an operating system with CPU contention prediction, ETR, a Java runtime that notifies when requests will be stalled due to heap garbage collection, and ETLIB, a library that throws an exception when a lock cannot be required within the deadline. ("ET" implies end-to-end tail mitigation support.)

In building LIBROS, the main challenge is creating two new capabilities in resource layers: prompt cancellation notification and accurate delay prediction (§IV-VI). In our implementation, these capabilities are written in 2300, 1000 and 250 LOC in ETOS, ETR and ETLIB, respectively (will be open sourced). To make applications benefit from these capabilities, LIBROS exposes simple APIs, e.g., we modified Cassandra and MongoDB only in 120 and 50 lines, respectively. Our evaluation shows that LIBROS is faster by 5-70% starting at 90P (the 90^{th}percentile) than popular practices such as speculative execution and is only 3% slower on average compared to the "best" (no contention) scenario.

II. BACKGROUND AND MOTIVATION

This section provides a concrete example of tail latencies due to multiple resource contention and reviews the major solutions in the last decade.

A. Multi-Resource Contention

Let us consider the following scenario. As a request arrives to a server, it might face a CPU contention; it cannot be delivered to the server application until the OS allocates a CPU for the application [1–8]. When the application receives the request, and if the code is written in a managed language such as Java, the request is usually converted from byte stream to an object (e.g. "`R = new Request(bytes)`") which can stall if the runtime is garbage-collecting the heap [9–14]. If not, the application can continue to process the request which is usually done in a lock protected function (e.g. "`synchronized processReq()`") which can incur a long delay when a heavy background activity is currently holding the same lock [15–18].

Fig. 1a, illustrating a CPU contention, shows the latency CCDF (complementary/reverse CDF) of Cassandra requests when the server process is competing over the same CPUs with other processes. The "+Noise" line shows around 15% of the requests experiencing tail latencies (vs. the stable "Best" line without CPU noises). Fig. 1b shows the same requests being stalled, not by CPU noises, but by Java garbage collection (GC) roughly 8% of the time due to other bulk requests that (de)allocate memory intensively. Likewise, Fig. 1c depicts a contention when some of the requests have to wait for a lock held by a background thread. Finally, Fig. 1d shows that when all the contentions are compounded, it will cause a larger tail area (25%). In fact, we observe a compounded effect where GC takes longer time because it is contending with the other CPU noises. Given this observation, it is hard to guarantee extreme stable latencies in multi-tenant, multi-resource storage systems.

B. State of the Art

The last decade has witnessed many novel solutions proposed to tame the resource contention problem, which we classify into four general categories: application-level modification, speculation, replica selection and resource-level optimization (**Table I**). We review their pros and cons in five axes: "application simplicity" implies no intrusive changes to the application; "efficiency" denotes no extra load (i.e. no speculative backup requests); "reactivity" means rapid reaction to latency perturbation in millisecond windows; "resource layer (RL) simplicity" suggests no heavy changes made in the resource manager (e.g. no policy changes); and finally "multi-resource coverage" means successful tail mitigation across multiple resource layers.

APPLICATION-LEVEL MODIFICATION re-architects tail-prone applications with a better computation and data management. For example, numerous key-value storage designs have been proposed for reducing contention between user and management operations or for handling workload skew

and cache inefficiencies [15–18, 26–31]. As shown in Table I, while it is efficient and reactive and does not change resource-level policies, it requires application redesign and does not cover contention outside the application.

SPECULATIVE EXECUTION treats the underlying system as unchangeable and simply sends a backup request (speculative retry) after some short amount of time has elapsed [32–35]. Many user-level storage adopt speculation for its simplicity and end-to-end coverage, but it causes extra load (i.e. speculative retry after waiting for the P^{th}-percentile latency will lead to $(100-P)\%$ backup requests).

REPLICA SELECTION predicts ahead of time which replicas can serve requests faster, often done in a black-box way (ease of adoption) without knowing what is happening inside the resource layers [36–38]. This requires detailed latency monitoring and expensive prediction computation for increasing accuracy. Most of the time, the prediction is only refreshed sparsely (e.g. every few minutes) [39]. As a result, it is not reactive to bursty contention that can (dis)appear in sub-second interval.

RESOURCE-LEVEL OPTIMIZATION eliminates the tail-inducing causes in the resource layer, for example, with better thread/task scheduling [1–7], storage layer optimization [40–44], and GC optimization [9–14].

While they are powerful, they come with the cost of modifying resource-level policies (scheduling, allocation, etc.) and only cover contention in their respective resource layers (e.g., cutting the tail in the OS storage stack [43–46] does not help user-level storage in an end-to-end way).

III. LIBROS

We now introduce LIBROS, an ecosystem of contention mitigation supports from library, runtime and operating system layers.

A. Design Overview

As mentioned in the introduction, it is fundamentally challenging to maintain all the benefits of the aforementioned techniques if we attempt to solve the tail latency problem only entirely in the application or resource layers. Within the LIBROS ecosystem, both application and resource levels work hand in hand. **Fig. 2** illustrates the LIBROS ecosystem, showing how a request flows from client to server through multiple resource layers in the operating system, runtime and library. To achieve all the five goals in Table I, LIBROS consists of five essential elements.

A) AN END-TO-END STORAGE `request` ABSTRACTION (Fig. 2a) connects all the resource layers together in mitigating tail latency. Currently, operating systems, runtime and libraries are often oblivious to the end-to-end request context. They operate on abstractions such as streams, packets and functions that are hard to map to the notion of "user request" and its latency sensitivity. With the prevalence of interactive services, user request should be a first-class citizen. `request` acts as a unifying abstraction for scattered resource layers.

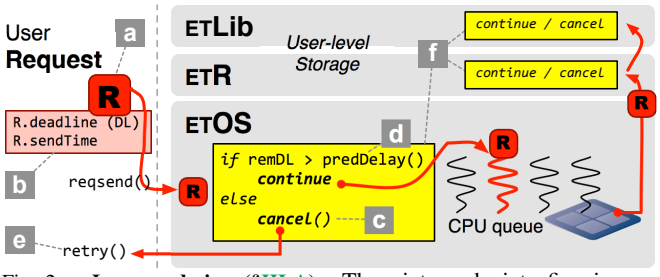

Fig. 2. **LIBROS design (§III-A).** The picture depicts five important elements of LIBROS: (a) end-to-end `request` abstraction, (b) deadline information, (c) request cancellation, (d) per-resource delay prediction, and (e) guided retry. Every resource layer adds (f) a uniform and stackable predict-server-or-cancel capability.

B) DEADLINE AWARENESS (Fig. 2b) is added to the resource layers through which `request` flows. The importance of deadline information has been highlighted many times [20, 47–50] and deadline-aware resources have been proposed such as in the TCP or block layer [43, 51, 52] but usually the awareness is only contained within the layer. With `request`, user's deadline information can be simply added and automatically forwarded across layers.

C) REQUEST CANCELLATION (Fig. 2c) is now supported in resource layers. That is, when replicas are available, user-level storage can tag requests as cancellable such that when the deadline cannot be met by a particular layer, the layer is given the liberty to cancel the request. The concept of cancellable tasks or requests is common in real-time community [53–55], but has not been fully deployed in many standard systems. Likewise, request cancellation is an essential mechanism that will notify applications that a particular resource in the datapath is currently contended. Google also promoted the efficiency of cancellable requests [56], but the article only described cancellation that is done in a user-level file system, while in our work, we advocate every major resource layer to be aware of deadlines and capable of cancelling requests.

D) PER-RESOURCE DELAY PREDICTION (Fig. 2d) can be now built in every major resource layer for making decision to serve or cancel every arriving request. The decision is made based on the request deadline and the current contention level inside the resource. The prediction ideally must be precise. Fortunately, a high precision is possible because the predictor is built inside the resource layer, hence has the full view of the contention inside the resource. For examples, ETOS is capable of measuring CPU contention, ETR of GC duration, and ETLIB of lock delays.

E) GUIDED RETRY (Fig. 2e) is now possible to be implemented by the application. Unlike timeout-based speculative retry that must wait for a certain time before sending backup requests, with LIBROS, the application can trigger retries as guided by the cancellation notification it receives from one of the resources.

With all the elements above, every resource layer can operate on a *uniform and stackable support* (Fig. 2f)—for

every request, predict the current delay, and cancel the request if the deadline cannot be met, or otherwise serve and forward the request to the next resource layer (if applicable). For every layer to know the remaining deadline since the request was sent ("`remDL`" in the figure), the `request` structure also contains the send time. This way, not all resource layers have to support prediction and cancellation, but only the often-contended major ones (more in §III-D).

B. Goals and Non-Goals

LIBROS design achieves all the goals in the following ways (as summarized in Table I). *(1)* The application remains simple because its job is straightforward: instantiate `requests` and perform speculative retries upon receiving cancellation notices. *(2)* Our approach is efficient because delayed requests are cancelled before being served, hence no extra load. *(3)* We achieve fast reactivity because the predictors we build always check the current delay, hence capable of adapting to millisecond burstiness. *(4)* We also maintain resource-level simplicity because we do not modify the original QoS policy, which is considered the most complex part of a resource layer, but rather we only need to build a predictor around it by reverse engineering the code. *(5)* Finally, we argue that fundamentally LIBROS provides sufficient coverage, especially when major resource layers (e.g. CPU/thread, memory, and disk management) adopt our method in supporting prediction and cancellation (for this reason we put "Major**" instead of a complete "$\surd$" in Table I).

We acknowledge that there are other sources of tail latencies (data skews, bad application load balancing policies, etc.). For this reason, applications should still enable timeout-based speculative retry to anticipate "unknown" root causes. Finally, we target replicated storage systems and assume scenarios where client and server machines live in the same data center (e.g. for data locality); data-center clock synchronization has been solved to nanosecond level [57].

C. APIs

The LIBROS ecosystem exposes simple APIs to applications. Due to space contraints, we decide to put API details to our supplemental material [58].[1] In short, we provide simple OS, runtime and library-level APIs such as `struct urequest`, `reqcreate()`, `reqsend()`, `reqrecv()`, and `notifyrecv()`, for sending user requests (along with deadline infromation) and receiving cancellation notifications.

D. Target Resource Layers and Applications

This paper shows the efficacy of LIBROS for major sources of contention in three resource layers: CPU contention in the OS layer, GC pauses in Java runtime, and synchronization delays in library lock functions. We target user-level, in-memory replicated key-value stores as the benefited applications. In building LIBROS, the largest challenge is in designing *prompt cancellation notification* and *highly precise delay prediction*. We show that while the overall "predict-then-cancel-or-serve" framework provides uniformity, simplicity and composability, every resource layer faces its own unique challenges to achieve these two objectives as described in the next three sections.

[1] We fully and completely acknowledge that that our extended report [58] will and should not be accounted in the review process. We provide a link to the report simply for interested readers who would like know more details.

IV. ETOS

ETOS is a Linux extension with request cancellation and prediction capabilities, focusing on CPU delays. Circumventing CPU contention by prediction and cancellation is an interesting challenge—how can an application detect that it is delayed in serving the arriving request while the application itself needs the CPU? Fortunately, today's latency-sensitive services form a client-server communication where requests pass through the OS (or some runtime), hence allowing ETOS to carry the burden of prediction and cancellation. We detail the challenges in implementing this feature in the TCP stack.

A. Cancellation Mechanism

There are two issues to tackle in cancelling requests: when to promptly cancel requests within the TCP stack and how to correctly remove requests from the TCP byte stream without breaking TCP semantics.

PROMPT CANCELLATION: We found two choices: when the packet arrives in the interrupt context (but before the TCP protocol processes the packet) or in the TCP receiving/processing context (which only happens when the destination process gets a CPU). The former is ideal for prompt cancellation and notification but it is not safe to interfere with the packet stream outside the regular TCP procedure (TCP packet processing is important for checking corruption, out of order delivery, and many other purposes). On the other hand, the latter is safer but slower.

A more ideal scenario is to get both of the benefits—process the packet during interrupt context such that cancellation can be done without any delay, which is the choice we made. This did not just involve moving Linux `tcp_recv()` function to the interrupt context, but we had to reroute the stream from TCP "prequeue" to the main TCP receive queue. One concern was that the interrupt context becomes more heavyweight, however we did not see the implication in our tests. Another fortunate news came nine months later when Linux developers also removed TCP prequeueing, but for a different reason; TCP prequeueing is optimized for "single process with blocking read" design, which is no longer a common style (polling-based calls are more frequent).

For cancellation notification, server-side ETOS sends back ACK messages that contain the IDs of cancelled requests (in general, `request` information is embedded in the TCP header fields in both directions). For prompt delivery, we disable

TCP delayed acknowledgement for cancellation ACKs, or otherwise they could be delayed by more than 40ms. Upon seeing the notice, the client-side ETOS passes up the message to the application via the `notifyrecv()` API.

CORRECT REQUEST REMOVAL: Cancelled requests should be treated differently from dropped (missing) packets. The former case implies that the packets have been received successfully, but not yet delivered to the application. Thus, cancelled requests should not alter the sequence numbers. However at the same time, we need to give the illusion that the cancelled requests have been read by the application such that the packet read ordering is not broken (specifically, Linux TCP's `copied_seq` variable should be updated carefully). To do this, we check the state of the application's receiving queue. If it is not empty, `copied_seq` should not be modified. We increase `copied_seq` accordingly until all packets in the receiving queue are read by the application.

B. CPU Delay Prediction

We now describe our prediction capability for measuring CPU delay on every latency-sensitive request within the CFS layer, Linux default and most complex scheduler. Unlike other works that modify thread/core management [1–6], our predictor does not modify CFS at all for adhering to the "resource simplicity" principle (§II-B). Our predictor only needs to change when CFS evolves, which rarely happens, e.g. within the span of Linux v4.0 to v4.20 (3.5 years), CFS only changes by 700 lines annually. For high precision, the predictor must consider many types of process/thread characteristics. Below, we describe our predictor from a naive version to a complete one (the pseudo-code is in [58]).

LINEAR PREDICTION: In a naive scenario where all process threads are CPU bound and long running on one CPU core and exists in the same user and priority group, the prediction can be based on a simple equation. For every thread T in the waiting (ready) queue, the future time slice when T will get the CPU is:

$$(\frac{T.vruntime - U.vruntime}{timeSlice} + 1) * timeSlice$$

where `vruntime` is the weighted time a thread has run on the CPU [59, 60], `timerSlice` is 4ms, and U is the next thread to be scheduled after T. Thus, to measure the CPU delay of a request designated to a thread X, we find all the threads supposed to be scheduled before X (the threads on the left side of X's position in CFS `rbtree`), then calculate how much time each must wait, and finally sum them all.

HIERARCHICAL PREDICTION: CFS however implements a hierarchy. When the scheduler picks a task to run next, it first searches from the top-level scheduling "entities" and takes the one with the lowest `vruntime`. If the chosen entity is not a real thread but rather another high-level scheduling entity (i.e. a nested hierarchy), the scheduler dives into it, searches through its runqueue, and repeats the procedure again until an actual thread is found. The chosen thread will be given a time slice to run (4ms) before being preempted. After this, the thread's runtime statistics are updated and also propagated up the hierarchy so that its new `vruntime` is properly reflected to the `vruntime` of its parent entities.

The implication of this hierarchy is that linear prediction no longer suffices. We must "simulate" what likely will happen, but at the same time not tamper with the actual accounting values. Thus, our predictor maintains a *shadow copy* of the entire hierarchy. When a prediction is needed, the shadow copy is first refreshed from the original values, after which the delay prediction is run on the shadow copy.

PRECISE TIMESLICE ADJUSTMENT: CFS performs scheduling on every timer interrupt or when a thread relinquishes CPU (e.g. when calling a blocking operation). Thus a thread does not necessarily run at a time slice boundary. Upon a timer interrupt, if the last execution time window of the currently running thread has not exceeded its assigned slice, CFS will skip scheduling on the current timer interrupt. Due to this imperfect time alignment, our predictor is occasionally off by roughly 4ms. Theoretically, if thread A starts at a timer interrupt t, then at the next interrupt $t+1$ we would intuitively assume that A has run for exactly 4ms. Upon further investigation, we found that the accounted execution time is slightly shorter than 4ms (e.g. 3.99ms). The reason is that the time taken for CFS to find the next running thread (e.g. 0.01ms) is not accounted into A's execution time. This causes imprecision when a thread's assigned slice is exactly (or multiple of) a time slice (4ms). With this observation, our predictor must slightly underestimate `vruntime` (e.g. by 0.01ms).

DEPENDENT AND I/O-BOUND THREADS: So far we assumed all threads are long running and independent, but this assumption does not hold for all types of workload. For example in the `vips` benchmark (an image processing system [61]), the threads are dependent on each other via synchronization primitives such as `futex`. For example, a thread A occasionally wakes up another mostly-idle thread B to execute some operation. Since B's CPU consumption is very small, it is favored by CFS to run next and B only runs in a short burst and then sleeps again. Our prediction is imprecise because of this nature of dependency and short burst that does not consume a full time slice.

To incorporate this behavior, ETOS marks dependent threads (via `futex` tracing) and estimates how long a dependent thread must wait before being wakened up by another thread. We record every dependent thread's `vruntime` and idle duration and use an exponential moving average to make the estimation. Let z_n denotes our estimation of a thread's idle duration at the n^{th} time it is waiting and t_n the real idle duration, then by using an exponential moving average, our estimation for the next $(n+1)^{th}$ wait time is:

$$z_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot z_n$$

Here α represents a decreasing weight, a constant smoothing factor between 0 and 1; a higher α will value more recent observations. We use 0.05 for α.

The same observation and technique can be made for I/O bound threads that often run short CPU bursts and wait for I/Os. However, note that when the application makes an I/O, it is not blocked when the I/O is served by the buffer cache or destined to a memory file system. Thus, we need to trace the actual device I/O time for recording t_n.

V. ETR (RUNTIME)

ETR ("R" stands for runtime) is a Java Virtual Machine (JVM) extension with request cancellation and delay prediction mechanisms pertaining to garbage collection (GC) pauses where application threads must be paused. While the literature in this area mostly focuses on manual GC tuning or GC optimization [9–14], ETR's follows the principle of "let GC stop the world, but let it not stop the universe" (e.g. in Taurus [12], a centralized manager that manages GC contention in distributed machines by rotating their GC periods). ETR however does not require a centralized manager and runs on independent instances. ETR can work with any stop-the-world GC algorithms because its main functionality is cancelling paused requests such that the higher-level distributed storage can continue retrying the requests somewhere else (i.e. do not stop "the universe"). Below we describe our solution to two main challenges: how the runtime can send cancellation notification when all application and many runtime threads are paused and how to predict GC pause delays.

A. Cancellation Mechanism

For the cancellation mechanism, we tried several metrhods from a naive to a more robust one. We tried modifying the Java I/O library but found issues with Java safepoints and also created a runtime-level thread similar to GC threads but found issues with SIGSEV signals when we unpark them, which all show the complexity of modifying a full-fledge JVM. More details are in [58]. Finally and successfully, we create a new runtime thread that does not have references to the part of the address space being reshuffled (outside the Java heap that contains application objects including `URequest` and `SocketImpl`). Our thread must know the sender's file descriptor for sending cancellation notices, but because this information resides in the Java heap (inside `SocketImpl`), we modify the JVM to copy relevant information that ETR needs and put them outside the Java heap.

B. GC Pause Prediction

JVM provides three GC algorithms, Parallel GC [62], G1GC [63], and ZGC [64]. Regardless of the implementation detail, we model GC execution time as a linear relationship to the number of live objects in the object graphs. Others have modeled GC in a linear way as well [65] but they model *when/how often* GC will take place, while we model *how long* every GC will pause. Because copying is the main bottleneck, our linear model is:

$$T_{gc} = \frac{N_{liveobj} \times T_{copy}}{N_{gcw}} + T_{ovh}$$

Here, T_{gc} is the predicted delay, $N_{liveobj}$ is the number of live objects, T_{copy} is the average copy time per object, N_{gcw} is the number of GC workers, and T_{ovh} is an additional constant overhead. As ETR has visibility on N_{gcw} (a constant configuration value) and $N_{liveobj}$ (after the fast initial traversal), we only need to profile the values of T_{copy} and T_{ovh}, which are dependent on the memory speed and other environmental factors. We tried several linear modeling algorithms such as RANSAC and OLS and found that RANSAC leads to the highest precision in our benchmarks. It successfully models T_{copy}, which depends on object sizes and memory copy speed, and T_{ovh}, which depends on some constant overhead, e.g. finding live/dead objects in mostly-static GC roots such as `ClassLoader`, `System Dictionary`, `JNI handles` and `Management Beans` objects graphs that might fluctuate in the beginning but will remain stable as the application runs for some time.

ETR's GC prediction is also not devoid of imprecision. Luckily ETR does not have to consider a wide range of application behaviors as ETOS does. ETR only needs to predict GC delays *within* the target application (e.g. the storage server), but *not across* different applications. In our case, we found that the memory usage pattern of simple key-value (de)allocations leads to a more predictable GC time compared to the more complex behavior of memory-intensive benchmarks.

VI. ETLIB

Finally, ETLIB advocates extending library functions that manage resources, such as locks, to expose more information that can help applications mitigate tail latency. In some cases, designing better locking strategies helps [31, 66–68], but for the simplicity argument (§III-B), ETLIB modifies neither lock internals nor its usages. For Cassandra, we apply this principle to the Java `ReadWriteLock` class used by Cassandra foreground and background functions. We believe that applying wrappers to other resource-managing library functions can be done in a similar way.

DELAY PREDICTION: The lock function is blind of the code that the lock protects, thus has no knowledge on how long the lock is being held. A naive way is to look at the call stack and take the average duration of the lock held by every function. For example, if the lock is held by a background operation, e.g. `rebalanceRing` or `flushTable` in Cassandra, the average duration is usually much higher than the lock time for foreground requests. Unfortunately call stacks and function names do not relay enough information.

We found that the dominant factor of locking duration is the number of for-loop iterations in the protected critical sections; for example, `rebalanceRing` contains a triple-nested loop iterating on N items that must be rebalanced.

Estimating this delay can be done within the library function or in a wrapper. We chose the latter for simplicity, by wrapping the `ReadWriteLock` functions. First, our application-specific wrapper records which function is currently holding the lock. With this information, we know the dominant factor and the code complexity of the function holding the lock, hence can estimate the delay more precisely by tracking historical information and modeling and projecting a simple average. For example, our wrapper can estimate that `rebalanceRing` of 32 and 256 nodes take around 25 and 130 milliseconds on average, respectively.

CANCELLATION: With the delay estimation available, our `ReadWriteLock` wrapper takes the deadline information from `URequest` and if the remaining deadline cannot be met, we return a delay "error." Neither the `ReadWriteLock` nor our wrapper needs to send the cancellation notice to the sender because distributed storage is usually already equipped with error propagation and retry mechanisms.

VII. IMPLEMENTATION

LIBROS is implemented in around 3550 LOC (ETOS in 2300 lines in Linux 4.10, ETR and ETLIB in 1000 and 250 lines in OpenJDK8, respectively). For the application, we modify Cassandra v3.11.6 [69] and MongoDB v3.3.12 [70] only in 120 and 50 LOC respectively, demonstrating the non-intrusiveness of our approach. Due to space constraints, the details can be found in [58].

VIII. EVALUATION

Our evaluation is primarily broken into two parts, performance (§VIII-A) and precision (§VIII-B).

CLUSTER SETUP: We use Emulab d430 machine [71] that has 16 cores (32 logical), 64 GB DRAM, and 1 Gbps network. The retry overhead (machine-to-machine ping-pong) is only 120µs. We deploy LIBROS on a small to large (20-node) clusters, half used as client and half as server nodes.

APPLICATION BENCHMARKS: We mainly use Cassandra and store the data in memory, replicated in 3 nodes. We use a microbenchmark where every client node sends 10,000 requests per second and a macrobenchmark with various load, noise, and read/write distributions. The experiments are performed several times to ensure reproducibility.

CONTENTION (NOISE) BENCHMARKS: For CPU noises, the Cassandra servers are colocated with CPU-intensive jobs. We use 3 CPU benchmark suites, Minebench [72], PARSEC [73] and SPLASH 2 [74], consisting of a total of 29 benchmarks. For GC pauses, we periodically send a large batch of non-critical requests to trigger GC within the experiment (e.g. mimicking a large database update or scan that is not latency sensitive, hence treated as regular, non-cancellable requests). For measuring the precision of our GC prediction, we use 3 memory-intensive benchmarks, SPECjvm2008 [75], DaCapo [76], and Renaissance [77]. For lock contention, we trigger `rebalanceRing`, an operational protocol in Cassandra.

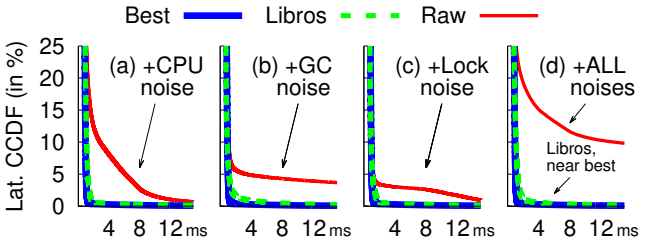

Fig. 3. **Single- to multi-layer tails (§VIII-A1).** The figures show CCDF graphs of the client-perceived latency distribution with (a) CPU noises, (b) GC pauses, (c) lock contention, and (d) all combined. In each graph, we show the "Best" (no contention), LIBROS, and "Raw" (no mitigation) lines.

TECHNIQUES EVALUATED: We evaluate LIBROS against popular practices such as Cassandra's replica selection [78] and speculative execution [79] with various timeout values. For example, "95P speculative retry" is often suggested [56, 80] where a backup request is sent after the 95^{th}-percentile latency value (the timeout value) has elapsed.

A. Performance Evaluation

For performance evaluation, we gradually evaluate LIBROS from a small to larger cluster setups and from LIBROS-vs-"raw" evaluation to LIBROS vs. other approaches.

1) Single- to multi-resource contention (3 nodes, synthetic benchmarks): We first repeat our motivational experiment (§II-A) and compare the "**Raw**" setup (*no tail mitigation*) with ETOS, ETR, ETLIB, and all combined. In this configuration, we use 1 client node and 2 server nodes where one of them experiences contention, and the injected contention is synthetic noises. For CPU noises, we insert 5 CPU-intensive threads per core; for GC pauses, we add a non-critical batch of writes (70 MB/sec) that will trigger GC; and for lock contention, we trigger a background operation that competes on the same lock with the client requests. We make the client choose the contended node first. The purpose is to show a "best-case" scenario that LIBROS can obtain.

First, the "**Best**" line in **Fig. 3** shows the latency CCDF of the client requests when there is *no contention* in the three resource layers. The line is vertically straight around x=0.7ms, the best-case scenario we should target.

Second, the **"Raw"** lines (*with noise*) in Fig. 3a-c show that the noises individually inflict long tail latencies to the user requests between roughly 5 to 10% of the time compared to the "Best" line. In Fig. 3d, all noises combined cause a larger latency tail about 25% of the time.

Finally, the LIBROS lines show that each of our methods can quickly react to the individual contention (Fig. 3a-c)

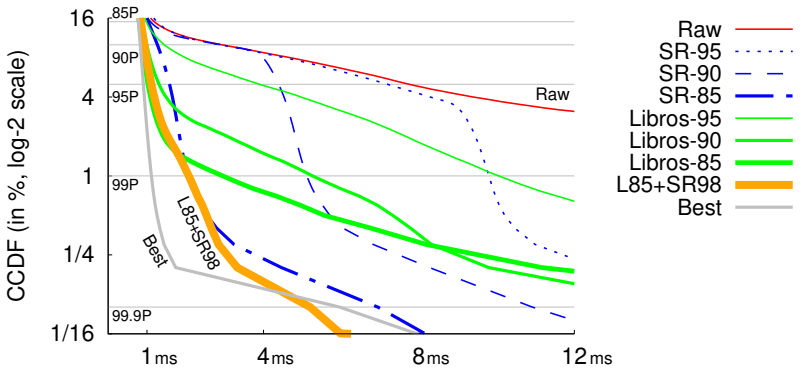

Fig. 4. **LIBROS vs. other techniques (§VIII-A2).** The latency CCDF (with **$\log_2$-scale y-axis**) compares LIBROS versus other popular techniques (please see §VIII-A2 for explanation).

TABLE II
Average latencies (§VIII-A2).

Techniques	**Average Latency** (ms)	***Slowdown vs. Best*** (in **%**)
Raw	2.17	234
SR-95	1.39	114
SR-90	1.05	62
SR-85	0.75	15
LIBROS-95	1.14	76
LIBROS-90	0.81	24
LIBROS-85	0.72	10
L85+SR98	**0.67**	*near best* $\rightarrow$ **3**
Best	**0.65**	—

as well as to the combined, multi-layer contention (3d). LIBROS successfully eliminates the tail latencies caused by the contentions (the large gap between the LIBROS and "Raw" lines). Compared to the "Best" line, LIBROS *is only 28% and 81% slower than the best case at 5% and 1% of the time (y=5 and y=1), respectively*.

We note that both in the "Best" and LIBROS lines, we still observe a small <1% latency tail, caused by "unknown" cases not covered by LIBROS (§III-B). For example, in the Emulab testbed, we always observe 0.3–0.5% long latency tail in a simple ping-pong workload, probably caused by network contention. This small residual tail can be amended by combining LIBROS with a small number of timeout-based speculative retries, as we show later.

2) LIBROS vs. other techniques (6 nodes, representative CPU benchmarks): We now thoroughly compare LIBROS with other popular practices. For a clean observation, here we only add CPU noises. We use three client nodes and three servers and every key-value is replicated three times. All the server nodes are shared between Cassandra and other CPU-intensive jobs. Here we use real representative CPU noises—for every core, we repeatedly run 2 threads of random CPU benchmarks (from the 29 benchmarks) in short bursts and short pauses in between every two runs.

Fig. 4 shows the same CCDF plot as in the prior figure, but now we have a *$\log_2$ y-axis* and 9 lines representing the best and raw scenarios, replica selection, and speculative retry and LIBROS with 3 different deadline/timeout configurations (85P, 90P and 95P values).

BEST AND RAW SCENARIOS: Let us quickly review the best and raw scenarios (left-most vs. right-most lines). Here we can see that the CPU noises introduce tail latencies roughly 15% of the time (y=15 in Fig. 4).

REPLICA SELECTION ("REPS"): Cassandra employs replica selection called "dynamic snitching" [78]. We find this method performs poorly in our benchmarks, almost near the raw scenario (not shown to not overcrowd Fig. 4). Because all the servers experience contention that (dis)appear in short intervals, Cassandra's snitch treats them as equally fast (or slow) and is incapable of declaring which server is contended in sub-second intervals.

TIMEOUT-BASED SPECULATIVE RETRY ("SR-85" TO "SR-95"): To recap, timeout-based speculative retry implies that if a request has not received a response after a certain duration (the timeout value), this method will send a backup request to another replica. In this technique, setting a proper timeout value is difficult. Too short means too pessimistic (leads to more inefficiency from sending too many backup requests). Too long means too optimistic and it will reduce application reactivity in retrying promptly. For fairness, we tried three timeout values, 1.5ms at 85P, 3ms at 90P, and 8ms at 95P (based on the percentile values in the Raw distribution).

Fig. 4 shows that speculation is effective in cutting tail latencies, but not as reactive as LIBROS, mainly because speculation does not know what is currently happening in the servers. For example, in the "SR-90" case, the speculation only sends backup requests after *waiting for 3 ms* (the 90P Raw value). On the other hand, LIBROS will quickly send a cancellation notice when it knows the request deadline cannot be met. Hence, our LIBROS-supported application does not have to wait for any timeout; cancellation notices were sent promptly and the application reacts much faster.

LIBROS: We can see that overall LIBROS is more efficient. Just like timeout-based retry, LIBROS depends on the application to provide the deadline value for the requests. Thus, we have three LIBROS lines in the figure with 85P, 90P and 95P deadline values for "apple-to-apple" comparison to speculative retries. LIBROS-85 and LIBROS-90 exhibit shorter tail latencies while LIBROS-95 does not trigger many retries (due to the relaxed 8ms deadline).

LIBROS WITH 98P (2%) SPECULATION: LIBROS only covers contention in the resource layers that participate in providing cancellation and prediction capabilities. It does not eliminate all sources of tails. As mentioned before, even the best-case lines always show almost 1% of tail latencies (due to network contention or other unknown factors). For

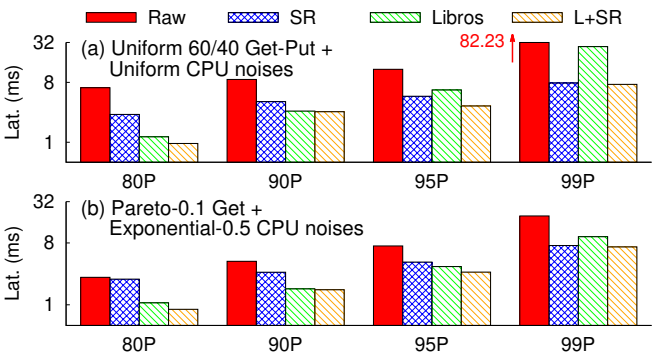

Fig. 5. **LIBROS with various load/noise distributions (§VIII-A3).** The bar graphs show the foreground request latencies, in $\log_2$ y-axis, at specific percentile values in x-axis.

this reason, it is wise to combine LIBROS with speculative retry but with an optimistic timeout. For example, in the **"S85+SR98"** line, the application sets an 85P deadline value for LIBROS and still sends backup requests when a 98P-value timeout has elapsed, sending only 2% backup requests.

OVERALL RESULTS: In addition to CCDF graphs, **Table II** shows the average latencies. As shown, cutting long latency tail by implication reduces the average latency. LIBROS with 85P deadline and with 98P speculative retry ("L85+SR98") is the most optimum because in the original distribution without tail mitigation (the Raw line in Fig. 4), the latency tail starts appearing around 85P, and in the LIBROS-85 line the residual latency tail is roughly 2%.

Overall, *compared to speculative retries* with the same P values, LIBROS reduces the completion time of user requests by 31-66% at 90P and 35-70% at 95P (see Fig. 4), and 5-23% on average (derived from Table II). *Compared to the best-case scenario*, our most optimum setup, "S85+SR98", is *only 3% slower on average* (see Table II) and 13% and 22% slower at 90P and 95P, respectively (see Fig. 4).

3) LIBROS *vs. speculative retry (20 nodes):* We now repeat a similar experiment but scale it up to 20 nodes (10 clients and 10 servers).

For **Fig. 5**a, we insert uniformly across all the servers 60% latency-sensitive `get` and 40% non-critical `put` operations (to trigger Java GC) as well as CPU noises similar to the previous section. The bar graph shows again that LIBROS combined with a relaxed speculation (the "L+SR" bars) is the most effective (lower latencies across the 80P to 99P values). At x=95P, speculative retry is slightly better than LIBROS (by itself) because LIBROS's predition is slightly imprecise (which we dissect in the next section). At x=99P, LIBROS latency is also high because, as mentioned before, we need speculative retry to mitigate the unknown noises (hence, "L+SR" is more effective).

For Fig. 5b, across the servers, we now spread the `get` operations with a Pareto α=0.1 distribution and the CPU noises with an Exponential λ=0.5 distribution; every server

TABLE III
Benchmark mixes (§VIII-B1). BENCHMARK MIXES FOR MEASURING ETOS' PREDICTION ACCURACY.

Mix	Description
A	User 1: blackscholes, swaptions, fluidanimate
	+ User 2: faceSim, freqMine, raytrace
B	barnes, fft, fmm, lu-cb, ocean-cp, radiosity, vips
C	radix, raytrace, volrend, water-nsquare, barnes, fft, dedup

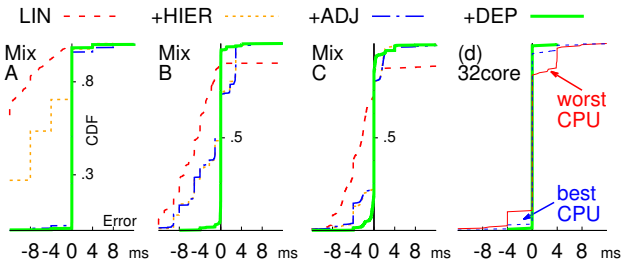

Fig. 6. **ETOS precision (§VIII-B1).** The figures show CDFs (0.0 to 1.0 in y-axis) of CPU delay prediction errors (D_{err} in ms in x-axis) across CPU benchmark mixes.

now observes a different request load and CPU noise distribution compared to its peers. The figure shows the same conclusion that while LIBROS outperforms speculative retry below 99P, the combined approach ("L+SR") always leads to the most superior performance.

B. Precision

The job of every predictor is to estimate how long an arriving request must wait in the corresponding layer. We now measure the precision of our three predictors.

1) CPU delay prediction (in ETOS*):* To measure ETOS' prediction precision, we use applications from the 3 CPU benchmark suites mentioned earlier. We start with a 1-core evaluation where we run a mix of 3 to 7 applications, as listed in **Table III**. We performed many mixes and repeated several times, but for space, we only present three mixes of benchmarks with unique results that show the importance of ETOS prediction features.

We measure ETOS imprecision by instrumenting Linux and adding information about when threads are running and preempted. In every 100 ms, we pick a random thread T in the wait queue and let ETOS predicts how long T has to wait before obtaining a CPU, i.e. the estimated delay (D_{est}). At the same time, our instrumentation also monitors the actual delay (D_{real}). We measure the error (D_{err}=D_{est}−D_{real}) and collect 1000 data points for every mix.

Fig. 6a-c show the CDF of the D_{err} data points in the three mixes. In every figure, we show the naive feature to the most complete feature (§IV-B): linear prediction (LIN), plus hierarchical prediction (+HIER), plus timeslice adjustment (+ADJ), and plus dependent thread awareness (+DEP).

In Mix-A (Fig. 6a) with CPU-bound benchmarks, our hierarchical feature (+HIER) is better than the linear prediction (LIN), but not too precise, until the timeslice adjustment is

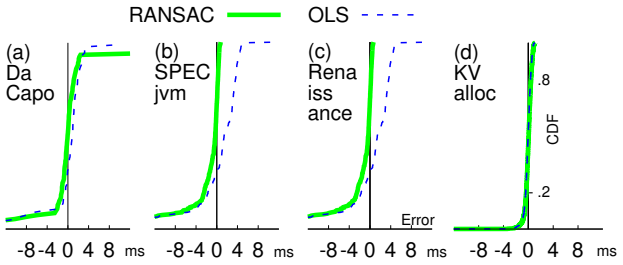

Fig. 7. **ETR precision (§VIII-B2).** The figures show CDFs of GC pause prediction errors (D_{err}), similar to Fig. 6.

added (+ADJ). In Mix-B (6b), with the presence of `vips`, an image processing system with dependent threads, the first three features are not enough, but with dependent thread awareness (+DEP), the predictor becomes more precise. Mix-C is a different mix but with `dedup` which also contains dependent threads, hence +DEP shows more precision. Overall, Fig. 6a-c show that ETOS is highly precise; the +DEP lines hover around x=0 (i.e. 0ms errors) and are only off by mostly $^{+}/_{-}$8ms in 5% of the time.

Finally, we show an experiment on 32 cores (Fig. 6d) where we ran 2-3 random benchmarks per core with a mix of 70% real CPU and 30% synthetic benchmarks (the memory space is not enough to run all real CPU benchmarks). The solid line in Fig. 6d shows that ETOS exhibits a 92% precision. The dashed "worst-cpu" line represents the CPU core where ETOS predicts accurately only 74% of the time. However, on the "best-cpu" core, ETOS is precise up to 98%. While being not fully precise, ETOS only mispredicts by 1-2 timeslices ($^{+}/_{-}$ 4-8ms across the x-axis).

2) GC pause prediction (in ETR*):* To evaluate ETR's precision, we chose 3 widely used Java benchmark suites, SPECjvm2008, DaCapo, and Renaissance. To measure imprecision, similar as above, ETR predicts how long every GC activity will run (D_{err}) and we also instrument the JVM to measure the actual GC duration (D_{real}). For every benchmark, we run the experiments until it captures 1000 D_{err} data points.

Fig. 7 shows the distribution of the D_{err} values. Every figure shows the two modeling algorithms we used (§V-B), RANSAC and OLS. In Fig. 7a-c, ETR is not precise in predicting GC pauses in benchmarks with complex memory usage, due to the reasons described before (§V-B). Fortunately, as we target storage systems, we notice that its memory usage pattern is not too complex (i.e. simple key-value (de)allocation). For now, it is sufficient for ETR to estimate under this simple pattern. Fig. 7d shows that with 1 MB of key-value (de)allocation per second with random sizes from 0.1 to 1 KB, ETR is only imprecise by $^{+}/_{-}$4ms in about 8% of the time with RANSAC.

3) Lock wait prediction (in ETLIB*):* For ETLIB evaluation, we use Cassandra's rebalance-ring background operation ("`calculatePendingRanges`") that can hold a `ReadWriteLock` that is also needed by foreground requests. In pure, masterless P2P system such as Cassandra, rebalancing is not a single iteration; since every node keeps sending the new view of the ring on every gossip, all the nodes will perform many iterations of `rebalanceRing` that can consume 35 to 150 ms depending on how many items to rebalance in the function's triple-nested loop. Thus, when a server node is locked in this function, user requests should be retried to other replicas.

Our Cassandra-specific lock wrapper (§VI) knows when the lock is held by `rebalanceRing` and also the variables that the loops are iterating on. With this, we can profile and keep a projection of lock wait time based on average values. For example, Fig. 8 shows a whisker plot where the x-axis is the number of nodes to rebalance and the y-axis how long the lock is being held. Every whisker bar represents five values: the 1st percentile (1P), first quartile, median, third quartile and the 99th percentile (99P). Lock unavailability depends on the number of nodes to rebalance, but for each node count, the average is a reasonable representative value to use for prediction.

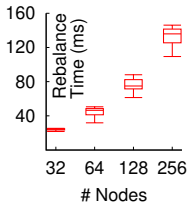

Fig. 8. **Lock time.**

Overhead and Additional Evaluations: We also have integrated LIBRO̅S with MongoDB (see [58]). • LIBRO̅S CPU overhead is under 1%, most of them come from ETOS because ETR and ETLIB do not need to run on every request. • Our performance evaluation shows that there is no performance instability due to cancellation. In essence, the number of requests retried with LIBRO̅S is similar to that of timeout-based speculative retry.

IX. CONCLUSION

We have demonstrated how an ecosystem of layered tail mitigation supports from library, runtime and operating system layers can "cut the tail together," resulting in a more effective outcome. LIBRO̅S integration with three major layers are a proof of concept how "`request`" can flow in an end-to-end manner and becomes a first-class concern and how request cancellation and delay prediction capabilities can be built in uniform and stackable manners. In our extended report [58], we extend further discussions including possible extensions to other resource layers, other notable contentions, and rooms for optimization.

X. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their tremendous feedback and comments. This material was supported by funding from NSF (grant Nos. CNS-1526304, CNS-1405959, CCF-2028427, and CCF-2119184) as well as generous donations from Facebook Faculty Research Award, Google Faculty Research Award, NetApp Faculty Fellowship, and CERES Center for Unstoppable Computing.

REFERENCES

[1] A. Daglis, M. Sutherland, and B. Falsaf, "RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs," in *ASPLOS '19*.

[2] C. Delimitrou and C. Kozyrakis, "Bolt: I Know What You Did Last Summer... In the Cloud," in *ASPLOS '17*.

[3] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley, "Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services," in *ASPLOS '15*.

[4] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazieres, and C. Kozyrakis, "Shinjuku: Preemptive Scheduling for µsecond-scale Tail Latency," in *NSDI '19*.

[5] J. Leverich, C. Kozyrakis, and J. Leverich, "Reconciling High Server Utilization and Sub-millisecond Quality-of-Service," in *EuroSys '14*.

[6] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads," in *NSDI '19*.

[7] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks," in *SOSP '17*.

[8] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding Long Tails in the Cloud," in *NSDI '13*.

[9] R. Bruno, D. Patricio, J. Simao, L. Veiga, and P. Ferreira, "Runtime Object Lifetime Profiler for Latency Sensitive Big Data Applications," in *EuroSys '19*.

[10] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro, "A study of the Scalability of Stop-the-World Garbage Collectors on Multicore," in *ASPLOS '13*.

[11] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen, "NumaGiC: a Garbage Collector for Big Data on Big NUMA Machines," in *ASPLOS '15*.

[12] M. Maas, K. Asanovic, T. Harris, and J. Kubiatowicz, "Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications," in *ASPLOS '16*.

[13] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu, "Yak: A High-Performance Big-Data-Friendly Garbage Collector," in *OSDI '16*.

[14] K. Suo, J. Rao, H. Jiang, and W. Srisa-an, "Characterizing and Optimizing Hotspot Parallel Garbage Collection on Multicore Systems," in *EuroSys '18*.

[15] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores," in *USENIX ATC '19*.

[16] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman, "Rocksteady: Fast Migration for Low-latency In-memory Storage," in *SOSP '17*.

[17] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "KVell: the design and implementation of a fast persistent key-value store," in *SOSP '19*.

[18] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees," in *SOSP '17*.

[19] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter, "RobinHood: Tail Latency Aware Caching – Dynamic Reallocation from Cache-Rich to Cache-Poor," in *OSDI '18*.

[20] H. Kasture and D. Sanchez, "Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads," in *ASPLOS '14*.

[21] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be Fast, Cheap and in Control with SwitchKV," in *NSDI '16*.

[22] P. A. Misra, M. F. Borge, I. Goiri, A. R. Lebeck, W. Zwaenepoel, and R. Bianchini, "Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints," in *EuroSys '19*.

[23] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding," in *OSDI '16*.

[24] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs," in *FAST '17*.

[25] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, "The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments," in *FAST '16*.

[26] D. Didona and W. Zwaenepoel, "Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores," in *NSDI '19*.

[27] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing," in *NSDI '13*.

[28] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout, "SLIK: Scalable Low-Latency Indexes for a Key-Value Store," in *USENIX ATC '16*.

[29] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store," in *USENIX ATC '15*.

[30] W. Reda, M. Canini, L. Suresh, D. Kostic, and S. Braithwaite, "Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling," in *EuroSys '17*.

[31] M. Sadoghi, M. Canim, B. Bhattacharjee, F. Nagel, and K. A. Ross, "Reducing database locking contention through multi-version concurrency," *Proc. VLDB Endow.*, vol. 7, no. 13, 2014.

[32] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the Outliers in Map-Reduce Clusters using Mantri," in *OSDI '10*.

[33] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI '04*.

[34] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *OSDI '08*.

[35] R. O. Suminto, C. A. Stuardo, A. Clark, H. Ke, T. Leesatapornwongsa, B. Fu, D. H. Kurniawan, V. Martin, U. M. R. G., and H. S. Gunawi, "PBSE: A Robust Path-Based Speculative Execution for Degraded-Network Tail Tolerance in Data-Parallel Frameworks," in *SoCC '17*.

[36] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers," in *ASPLOS '17*.

[37] Z. Wu, C. Yu, and H. V. Madhyastha, "CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services," in *NSDI '15*.

[38] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz, "Wrangler: Predictable and Faster Jobs using Fewer Resources," in *SoCC '14*.

[39] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection," in *NSDI '15*.

[40] A. Gulati, I. Ahmad, and C. A. Waldspurger, "PARDA: Proportional Allocation of Resources for Distributed Storage Access," in *FAST '09*.

[41] A. Gulati, A. Merchant, and P. J. Varman, "pClock: An Arrival Curve Based Approach For QoS Guarantees In Shared Storage Systems," in *SIGMETRICS '07*.

[42] A. Gulati, G. Shanmuganathan, I. Ahmad, C. A. Waldspurger, and M. Uysal, "Pesto: Online Storage Performance Management in Virtualized Datacenters," in *SoCC '11*.

[43] M. Hao, H. Li, M. H. Tong, C. Pakha, R. O. Suminto, C. A. Stuardo, A. A. Chien, and H. S. Gunawi, "MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface," in *SOSP '17*.

[44] M. Hao, L. Toksoz, N. Li, E. E. Halim, H. Hoffmann, and H. S. Gunawi, "LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network," in *OSDI '20*, 2020.

[45] J. He, D. Nguyen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Reducing File System Tail Latencies with Chopper," in *FAST '15*.

[46] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Split-level I/O scheduling," in *SOSP '15*.

[47] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "GRASS: Trimming Stragglers in Approximation Analytics," in *NSDI '14*.

[48] N. Li, H. Jiang, D. Feng, and Z. Shi, "PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage," in *EuroSys '16*.

[49] H. Zhang, K. Chen, W. Bai, D. Han, C. Tian, H. Wang, H. Guan, and M. Zhang, "Guaranteeing Deadlines for Inter-Datacenter Transfers," in *EuroSys '15*.

[50] H. Zhu and M. Erez, "Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems," in *ASPLOS '16*.

[51] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP," in *SIGCOMM '10*.

[52] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better Never than Late: Meeting Deadlines in Datacenter Networks," in *SIGCOMM '11*.

[53] J.-J. Chen, T.-W. Kuo, C.-L. Yang, and K.-J. King, "Energy-Efficient Real-Time Task Scheduling with Task Rejection," in *DATE '07*.

[54] S. Jamin, S. Shenker, L. Zhang, and D. D. Clark, "An Admission Control Algorithm for Predictive Real-time Service," in *NOSSDAV '92*.

[55] S. Lauzac, R. Melhem, and D. Mosse, "An Efficient RMS Admission Control and its Application to Multiprocessor Scheduling," in *IPDPS '98*.

[56] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM (CACM)*, vol. 56, no. 2, 2013.

[57] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat, "Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization," in *NSDI '18*.

[58] "LIBROS Extended Report (mainly for interested readers and not expected to be counted in the review process)," https://tinyurl.com/ycsaysop or https://bit.ly/2SeH5qb.

[59] "CFS Scheduler," https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[60] "Linux Scheduler – CFS and Virtual Run Time (vruntime)," https://oakbytes.wordpress.com/2012/07/03/linux-scheduler-cfs-and-virtual-run-time/.

[61] "libvips - A Fast Image Processing Library with Low Memory Needs," https://libvips.github.io/libvips/.

[62] "Parallel Garbage Collector," https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html.

[63] "Garbage-First Garbage Collector," https://docs.oracle.com/javase/9/gctuning/garbage-first-garbage-collector.htm.

[64] "Z Garbage Collector," https://wiki.openjdk.java.net/display/zgc/Main.

[65] P. Libic, L. Bulej, V. Horky, and P. Tuma, "On the limits of modeling generational garbage collector performance," in *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2014.

[66] B. Jeong, A. Khan, and S. Park, "Async-LCAM: A Lock Contention Aware Messenger for Ceph Distributed Storage System," in *Cluster '19*.

[67] S. Kashyap, I. Calciu, X. Cheng, C. Min, and T. Kim, "Scalable and Practical Locking with Shuffling," in *SOSP '19*.

[68] B. Tian, J. Huang, B. Mozafari, and G. Schoenebeck, "Contention-aware lock scheduling for transactional databases," *Proc. VLDB Endow.*, vol. 11, no. 5, 2018.

[69] "Apache Cassandra," http://cassandra.apache.org/.

[70] "MongoDB Official Website," https://www.mongodb.com.

[71] "Emulab D430s," https://gitlab.flux.utah.edu/emulab/emulab-devel/wikis/Utah-Cluster/d430s.

[72] "NU-MineBench," http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html.

[73] "Princeton Application Repository for Shared-Memory Computers (PARSEC)," https://parsec.cs.princeton.edu.

[74] "SPLASH-2x Benchmarks," https://parsec.cs.princeton.edu/parsec3-doc.htm#splash2x.

[75] "SPECjvm 2008," https://www.spec.org/jvm2008/.

[76] "DaCapo Benchmarks," http://dacapobench.org.

[77] "Renaissance Suite," https://renaissance.dev.

[78] "Cassandra Snitches," https://docs.datastax.com/en/archived/cassandra/2.0/cassandra/architecture/architectureSnitchesAbout_c.html.

[79] "Cassandra Speculative Query Execution," https://docs.datastax.com/en/developer/java-driver/3.2/manual/speculative_execution/.

[80] "Tuning Speculative Retries to Fight Latency," https://www.youtube.com/watch?v=uRJSuQofJWQ, 2016.