Improving Tail Latency of Stateful Cloud Services via GC Control and Load Shedding

Daniel Fireman*†, João Brunet†, Raquel Lopes†, David Quaresma†, Thiago Emmanuel Pereira†
* *Instituto Federal de Alagoas*, Maceió, Brazil
danielfireman@ifal.br
† *Universidade Federal de Campina Grande (UFCG)*, Campina Grande, Brazil
{joao.arthur,raquel,temmanuel}@computacao.ufcg.edu.br, david.quaresma@ccc.ufcg.edu.br

***Abstract*—Most of the modern cloud web services execute on top of runtime environments like .NET's Common Language Runtime or Java Runtime Environment. On the one hand, runtime environments provide several off-the-shelf benefits like code security and cross-platform execution. On the other hand, runtime's features such as just-in-time compilation and automatic memory management add a non-deterministic overhead to the overall service time, increasing the tail of the latency distribution. In this context, the Garbage Collector (GC) is among the leading causes of high tail latency. To tackle this problem, we developed the Garbage Collector Control Interceptor (GCI) – a request interceptor algorithm, which is agnostic regarding the cloud service language, internals, and its incoming load. GCI is wholly decentralized and improves the tail latency of cloud services by making sure that service instances shed the incoming load while cleaning up the runtime heap. We evaluated GCI's effectiveness in a stateful service prototype, varying the number of available instances. Our results showed that using GCI eliminates the impact of the garbage collection on the service latency for small (4 nodes) and large (64 nodes) deployments with no throughput loss.**

***Index Terms*—cloud-computing, tail-latency, garbage-collector, runtime, cloud-services**

I. Introduction

Managed programming languages, i.e., languages in which programs execute on top of a runtime environment, are prevalent in current cloud data-centers [1]. Companies such as Twitter [2] and Facebook [3], for instance, are writing most of their applications in Scala and PHP. At the same time, cloud platforms such as Google AppEngine [4] and Microsoft Azure [5] are explicitly targeting the execution of services written in managed languages.

These runtime environments (RTEs) are popular because they provide several off-the-shelf benefits. For example, RTEs typically provide automatic memory management, which removes the need for developers to manually track and release memory, avoiding entire classes of bugs. RTEs also enable cross-platform execution, allowing the same code to run the service on multiple operating systems.

However, some of these benefits come with a performance penalty. For example, it is well-known that the Garbage Collector (GC) negatively impacts the service performance due to CPU competition or stop-of-the-world pauses [6], [7]. To concretely illustrate the problem, let us analyze Figure I. It shows the latency empirical cumulative distribution function (ECDF) [8] and highlights the median and 99.9^{th} percentile for a synthetic constant CPU-bound workload processed by a Java™ [9] HTTP service instance. Although all requests are equal and the workload is uniform, the 99.9^{th} percentile of the latency is considerably worse than the median. Eliminate the tail latency caused by the GC's non-deterministic overhead is the primary goal of this work.

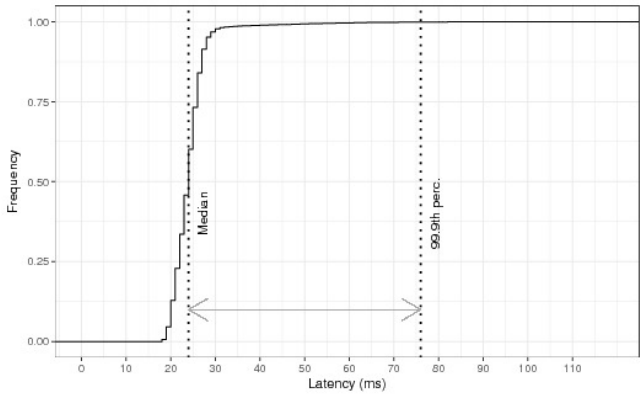


Fig. 1. Latency ECDF of a single experiment run (4 replicas, Java stateful service)

Since cloud services are typically structured as multi-tiered, large-scale distributed systems, serving a single request may involve requests to tens or hundreds of servers. For example, a query to Facebook's real-time data management system involves hundreds of servers [10] and some stages of a Bing search query may hit thousands of servers in parallel [11]. In this context, the non-deterministic overhead caused by the GC to the service time of an instance ends up lengthening the tail of the service latency distribution, as even temporary latency spikes from individual nodes may ultimately dominate end-to-end latencies [6], [11], [12].

One could argue that tuning the GC eliminates its impact on latency; however, it is a hard task. First, GC tuning depends on characteristics of the load the cloud service is subjected to, which can be very dynamic, particularly in worldwide distributed systems. Second, it depends on the service code, which can be deployed many times a day through continuous delivery pipelines. Another option would be to switch back to manual memory management, either by programming in

a language that does not have a runtime support (e.g., C, C++) or by completely turning off the garbage collector. That would lead the team to lose the productivity gains of managed programming languages [13]

In a previous work [14], we conducted a preliminary evaluation of the first version of our strategy to improve the long tail of the latency histogram. We were able to assess its feasibility in a single-instance service. The preliminary results demonstrated GCI efficacy and motivated the evaluation of GCI's usage for different types of services and with multiple replicas behind a load balancer. That led to significant changes in GCI's design and implementation.

In this paper, we present the new version of the Garbage Collector Control Interceptor (GCI), a practical approach to improving the tail latency of cloud services by avoiding latency spikes caused by non-deterministic garbage collector activity. Using Figure I as a reference, GCI's primary goals are to narrow down the distance from the median to the 99.9^{th} percentile line without significantly compromising the throughput of the service. Instead of attempting to minimize the impact of GC interventions, GCI transforms these interventions into temporary failures. It does that by controlling the GC and shedding the incoming load during the collection, avoiding CPU competition and stop-the-world pauses while processing requests. In practice, cloud load balancers would transparently route shed requests to available service instances.

GCI is entirely agnostic regarding the service internals and workload. Furthermore, GCI's usage neither requires manual configuration nor understanding of the runtime system's internals. To decide what is the best time to perform the garbage collection, GCI tracks the heap usage and the number of incoming and processed requests. Allied to that, it uses the available load shedding mechanism – i.e., Service Unavailable status code and Retry-After response header in HTTP services [15] – to advise load balancers to stop sending requests (and try another service instance). GCI executes in a completely decentralized manner and relies on only two RTE-specific API calls, making it easy to be implemented.

We implemented GCI for the Java programming language because it is the only managed language which does not allow developers to completely disable automatic memory management, making it more difficult to control the garbage collection. We evaluated GCI's effectiveness using a stateful service prototype, which could be a key-value in-memory store or a web server that keeps session information across requests. We started evaluating stateful services for two reasons: i) because of the importance of widely used services like Cassandra [16], Elasticsearch [17] and in-memory datastores like Netflix's EVCache [18] and Hazelcast [19]; and ii) because the heap usage pattern of those services leads to full garbage collections, which are the most impactful kind of collection. In order to assess GCI's suitability to cloud scale, we simulated small (4 nodes) and large (64 nodes) deployments. Our results showed that using GCI eliminates the impact of the garbage collection on the service latency for both deployments with no throughput loss.

We organized the remainder of this paper as follows. Section II describes the design and implementation of GCI. In Section III we describe our research questions and detail the simulated evaluation and results. In Section IV we summarize the simulator validation. In Section V we describe related work and finally, we present our final remarks in Section VI.

II. Garbage Collector Control Interceptor (GCI)

To help cloud service developers to deal with the impact of non-deterministic GC interventions on latency we proposed and implemented an easy-to-use technique called the Garbage Collection Control Interceptor (GCI). The overall design is based on a very common idiom in cloud services, the interceptor [20]. Instead of attempting to minimize the impact of garbage collector interventions, GCI transparently transforms these interventions into a temporary node unavailability. GCI controls the garbage collector and sheds the incoming load during the collection, which avoids CPU competition and stop-the-world pauses while processing requests.

The main advantages of using GCI instead of tuning the garbage collector or changing the code are that GCI is entirely agnostic of the service internals and transparently adapts to changes in the load. Its specification is independent of runtime and communication protocol. Furthermore, GCI is straightforward to use[1] as it requires no specific configuration or understanding of the runtime system's internals.

Figure 2 illustrates the overall architecture of GCI. It has two main components (highlighted in gray): i) the Proxy – a multiplatform, runtime-agnostic intermediary responsible for controlling the garbage collector and shedding the load – and ii) the Request Processor – a thin layer which runs within the service and executes two runtime-specific commands, i.e., checking the heap allocation and performing a garbage collection.

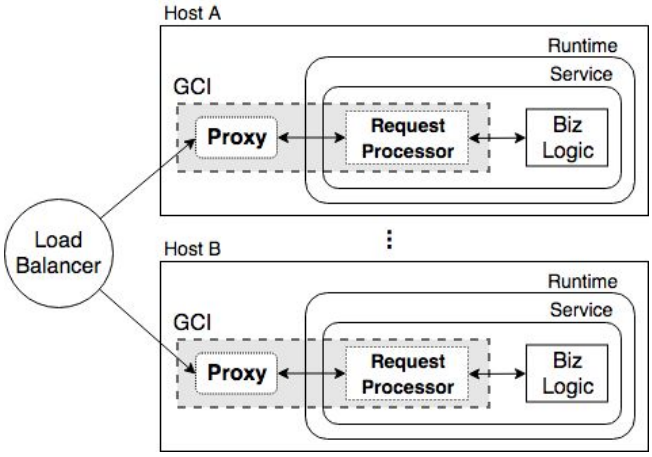


Fig. 2. GCI Architecture. The Proxy and the Request Processor intercept the execution flow of the Service. The Proxy controls the garbage collector and sheds the load. The Request Processor, which runs within the Service, is responsible for checking the heap allocation and performing garbage collection.

[1]Available at https://github.com/gcinterceptor

We next detail the core concepts of GCI. We first describe the Proxy, which is responsible for controlling the GC and shedding requests. Then we explain the Request Processor and how it receives commands and sends information to the GCI Proxy.

A. GCI Proxy

Figure 3 illustrates the general operation of the GCI. If the service is unavailable, because the garbage collector is already running, GCI Proxy sheds the request. Shedding a request means to inform the load balancer about the instance's unavailability (e.g., Service Unavailable HTTP response). That allows the cloud load balancer to resend the request to the next service instance.

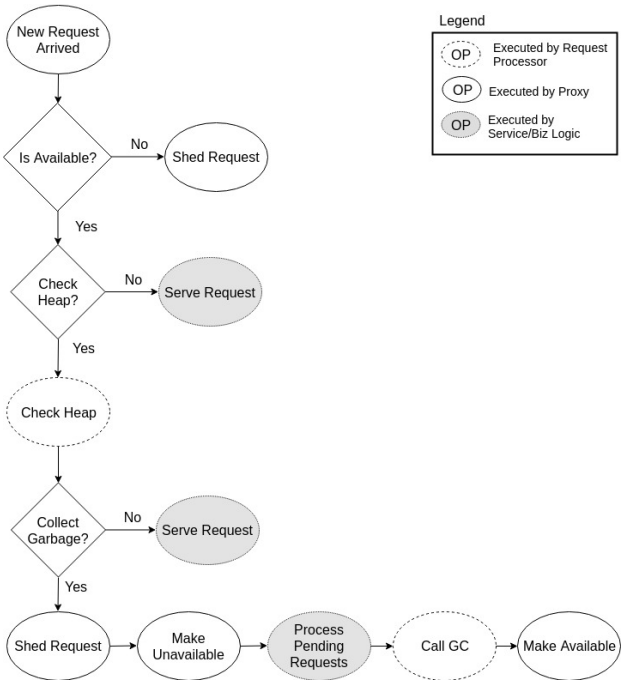


Fig. 3. GCI flow.

If the service is available, and it is time to check the heap usage, the GCI Proxy commands the Request Processor to do so. With such information about the heap, the Proxy decides whether it is time to clean up memory. If it is not yet time for the garbage collector to run, the usual request processing flow takes place. If it is time to collect the garbage, the GCI Proxy marks the service as unavailable and sheds the incoming requests until the service is available again. In addition to that, GCI Proxy tracks the completion of all pending requests. When the service is done processing all pending requests, the GCI Proxy triggers the heap cleanup, which is performed by the Request Processor. When the garbage collector finishes, the GCI Proxy marks the service as available.

The two critical aspects of GCI Proxy behavior are: i) when to check the heap usage, and ii) when to run the garbage collection. We next describe the trade-offs and choices behind those two decisions.

1) Deciding when to check the heap usage: Java, Python, Ruby and many other runtime environments export methods to access information about heap utilization [21]–[23]. In general, those are lightweight operations and return reasonable estimations. However, there are cases in which the determination of current memory usage is expensive, for example, when objects are not contiguously packed [21].

As GCI aims at adding minimum overhead, to check memory usage at each request would be inefficient, especially in highly loaded production deployments. The GCI checks the heap after processing N requests. The choice of N is a trade-off. If N is too small, the overhead of checking memory usage can become prohibitive. Otherwise, either we lose control over the garbage collector, or the system could run out of memory.

Defining N is not trivial. A fixed value for N may not always be efficacious, because requests could differ and the load could vary over time leading to different heap usage patterns. Furthermore, the usage of complex, computationally intensive prediction algorithms could negatively impact performance. To cope with these constraints, N starts with a conservative value (e.g., 10) and varies slowly during the service lifetime, according to the following algorithm:

The GCI Proxy keeps a cyclic buffer with the latest values of N (history) and tracks the number of requests processed between consecutive garbage collections. The history starts with the first value of N. After every collection, the number of requests processed is added to the history, and the minimum value in the history is selected to be the next value of N. Furthermore, there is an upper bound to the value of N as another way of avoiding the runtime system to trigger the garbage collection unexpectedly. Finally, as the arrival rate and the amount of memory needed to compute each request can vary a lot over time, keeping a long history might not improve the decision. Thus GCI keeps a small history that considers the N value reached for the last 5 controlled garbage collection executions.

2) Deciding when to run the garbage collector: For every N requests processed, GCI checks the heap. When the amount of heap allocated by the service reaches a certain threshold (T), GCI starts its preparation to collect garbage. Alike N, the value of T starts small (e.g., 30% of the heap or generation size) and increases after each collection until a maximum value.

As non-deterministic collections negatively impact the service latency, an essential aspect to consider is whether the target runtime allows disabling automatic runs of the garbage collector. As the collection control algorithm is meant to be runtime agnostic, it avoids spurious GC executions by setting the maximum value of T to a conservative factor of the heap or generation size (e.g., 70%). In case that is not enough, every uncontrolled garbage collection execution leads to a decrease of the value of T.

Finally, GCI runs in a completely decentralized manner, and that could lead to the case where all or most instances become unavailable at the same time, and thus the service would not be able to process requests. To avoid throughput loss, each

instance changes T by a small random value after controlled collections. Those updates keep happening throughout the service lifetime.

B. Request Processor

The GCI Request Processor is a thin layer which runs inside each service instance to execute two runtime-specific commands: i) check heap usage, and ii) perform garbage collection. For runtimes in which it is not possible to switch off the automatic collection, the GCI Request Processor also notifies the GCI Proxy in case of spurious garbage collector executions.

Using HTTP cloud services as an example, the communication between the GCI Proxy and the Request Processor uses HTTP requests with specific headers[2]. Responses and notifications come back to the Proxy via response headers. For instance, when it is time to perform a garbage collection, the Proxy sends the "GCI" request header with value "GC". Procedure 1 depicts the pseudocode of such control protocol.

Procedure 1 HTTP Request Processor

```
switch RequestHeaders.Get("GCI")
case "GC":
  Runtime.ForceGC()
case "ALLOC":
  Alloc ⇐ Runtime.GetHeap().GetUsage()
  ResponseHeaders.Put("GCI − ALLOC", Alloc)
end switch
return True
```

III. Evaluation

We conducted our evaluation to answer the following question: does GCI shorten the tail of the latency distribution without significantly penalizing the service throughput? To answer such question, we carried out simulation experiments to evaluate the GCI usage in a cloud service scenario.

Our experimental design followed a full factorial design with two factors and 5 repetitions. The metrics considered to evaluate the GCI performance were the latency distribution and the average throughput. One crucial factor defined the size of the cluster running the service, and we called this factor the cloud scale. The cloud scale is needed to properly evaluate the impact of the GCI on the latency when, by chance, the load balancer directs requests to unavailable nodes in a cascade. The cloud scale factor varied in two levels: small (4 nodes) and large (64 nodes). The other factor is the status of the GCI, which can be on or off.

We built a discrete-event simulator[3] as well as a workload generator that creates and sends requests. The simulation model consists of a load balancer that receives the requests and routes them to a set of nodes in a round-robin fashion. The load balancer transparently forwards shed requests to the next node. Each node has one chance to process the request, and if all the nodes were unavailable, the request fails. The simulated load balancer behavior mimics the Nginx, a widely-used load balancer [24], which has round-robin as the default load balancing method. A request is successfully finished if at least one node was able to handle it. In this context, we measured the latency of the successful requests at the load balancer and defined throughput as the rate of successful requests processed per unit of time.

In addition to the load balancing behavior, the simulator also models the behavior of the nodes that handle the requests and the resource consumption (heap usage) in these nodes. Instead of modeling these aspects directly, the simulated nodes replay the behavior of a real server, observed in a small-scale measurement experiment.

To carry out this measurement experiment we implemented a version of the GCI in compliance with Java version 10 [9][4] and a toy stateful service whose requests allocate 256KB of memory, compute 5 thousand prime numbers[5] and update its state. We used Nginx as the load balancer and its default balancing algorithm, round-robin. The service was executed in a virtual machine with two cores and 1GB of RAM. The Nginx, the GCI Proxy, and the load generator ran in a separate virtual machine, configured with four cores and 2GB of RAM. The cloud service JVM was configured to work in server mode, with a 512MB heap (50% of this heap is used to store the young generation) and to use the Garbage-First garbage collector (G1GC) [25] (default in Java 10 when running in server mode). The state allocated by the service was fixed and equals to 132MB, which is less than the quantity of heap left to the tenured generation. As an attempt to narrow the causes of latency variance to the garbage collection, we chose to send the same request at a constant and low rate, which was 30 requests per second. Each run lasts around 10 minutes, which is enough to reach the steady state after discarding the first 4 minutes of each test to minimize the effects of JVM warm-up [26]. We restarted the service JVM, the GCI Proxy, and Nginx before the next experiment run.

This experiment outputs a replay log with an entry for each request processed during the experiment. The entry has a status code, to indicate whether the request was served or shed, and the latency to process the request, measured in the load balance. For the simulation experiment, we did not repeat the replay logs generated by the measurement experiment. Thus, the simulation of a 64 servers scenario required the previous execution of 64 replicas the 1-instance measurement experiment, one for each simulated server.

Each simulated node replays its log sequentially, as it receives news requests from the load balancer. On receiving a request, there are two possible outcomes, based on the entry status code. When the status code indicates the request was successfully processed, the simulator registers the request latency as seen in the log entry. In its turn, when the entry

[2]A different GCI implementation, for another kind of service, could use another mechanism

[3]Se Section IV to see the validation process of the simulation model.

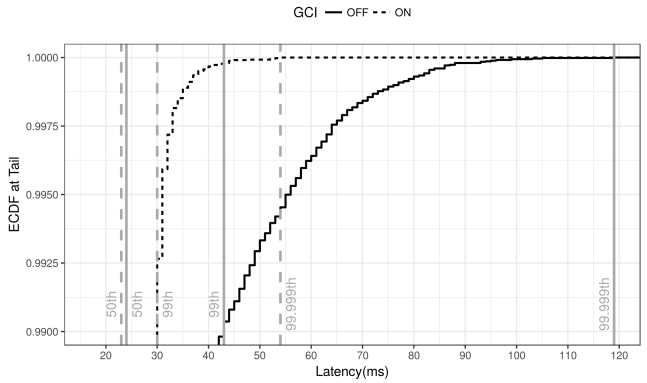
[4]Available at http://github.com/gcinterceptor/gci-java

[5]https://github.com/gcinterceptor/java-experiments/tree/master/msgpush

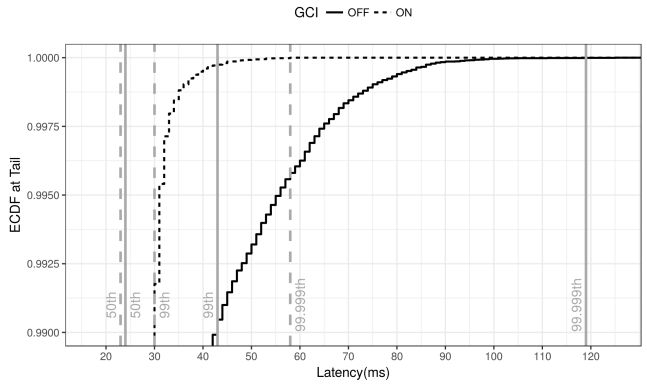
status code indicates the request was shed, the simulator denies the execution of the request under simulation and sends it back to the load balancer. In this late case, the load balancer keeps the response time of the shed request to compute the simulated latency and routes the request to the next node.

A. Results

Figure 4 presents the tail of the latency's empirical cumulative distribution function (ECDF) [8] for experiments with GCI on and off. These results indicate that by enabling GCI we decrease the tail of the latency distribution for both large and small stateful services.



(a) Small Service (4 nodes)



(b) Large Service (64 nodes)

Fig. 4. Latency ECDF at tail showing that enabling GCI improves the performance of small and large stateful services. The vertical-axis starts at the 99^{th} percentile. Vertical lines show the percentiles when GCI is on (dashed) and off (solid).

Enabling GCI for a small stateful service improves the 99^{th} percentile of the latency by 30%. The benefits are more significant as we move towards the end of the distribution, reaching 47% at the 99.999^{th} percentile. Furthermore, it is important to notice that enabling GCI not only made the latency better (moving towards the median) but also improved predictability by shortening the distance between the 99^{th} and 99.999^{th} percentiles by a factor of 3.

Table I details the performance of both service sizes. As our experiments increased load and resources in the same proportion, we expected very small or no difference in latency distributions. Our results confirmed that the GCI algorithm is ready for cloud scale, as large and small service deployments experienced very similar improvements. Furthermore, results also show that GCI's overhead is negligible, as medians are very close in both service types.

TABLE I
DETAILED VIEW OF THE TAIL LATENCY OF SMALL AND LARGE SERVICES. RANGES ARE THE MINIMUM AND MAXIMUM VALUES OF EACH STATISTIC ACROSS ALL EXPERIMENT REPLICAS. ALL VALUES ARE IN MILLISECONDS AND ROUNDED TO THE 3 MOST SIGNIFICANT DIGITS.

	Small Service		Large Service	
Latency Perc.	GCI Off	GCI On	GCI Off	GCI On
50^{th}	[24, 24]	[23, 23]	[24, 24]	[23, 23]
99^{th}	[43, 43]	[30, 30]	[42, 43]	[30, 30]
99.9^{th}	[74, 77]	[34, 35]	[75, 75]	[36, 36]
99.99^{th}	[90, 97.2]	[42, 42.4]	[85, 94]	[47, 49]
99.999^{th}	[97.9, 114]	[57.6, 58.3]	[111, 119]	[58, 58]

These results allow us to answer the research question by showing that GCI reduces the tail of the latency distribution across all considered scenarios. Besides, there is a substantial decrease in the distance between the 99^{th} and 99.999^{th} (tail variability), which leads to a more predictable service time profile.

Impact on throughput: GCI's load shedding mechanism did not lead to a throughput penalty in any of the considered scenarios. One could argue that sending the same request at a constant rate would necessarily lead to the unavailability of all serving nodes. The achieved result is a consequence of the efficacy of the randomized shedding threshold mechanism used. Without this mechanism, all the nodes receiving exactly the same load would need a garbage collection at the same time, which, in turn, would have decreased the throughput when the GCI was on.

In summary, regardless of the service size, enabling GCI substantially reduces not only the tail of the latency distribution but also the ratio of the tail to the median, leading to a more predictable latency profile. GCI's overhead is negligible, and there is no throughput loss for the considered cases.

IV. VALIDATION

Our simulation model was validated through measurement experiments. To carry out these measurement experiments we implemented the GCI, the workload generator and a stateful service prototype[6]. We executed the measurement experiments for validation by varying the factors presented in Section III, which are the GCI on/off and cloud service scale small/large. For the sake of simplicity, we ran the validation experiments for services with 1,2 and 4 service nodes. We replicated each system experiment test five times. We compared the results of the simulation and the measurement experiments in the same scenarios. This is the most reliable and preferred way to validate a simulation model [27]. Figure 5 shows the result of the comparison for validation.

[6]These implementations are available at https://github.com/gcinterceptor/gci-simulator

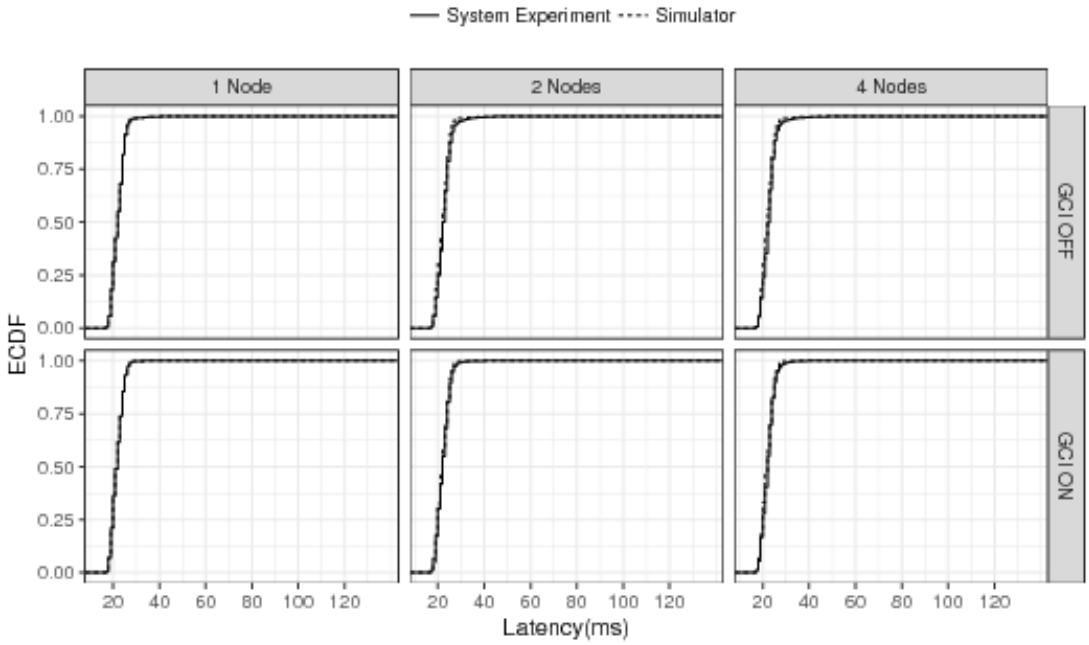


Fig. 5. Comparison of simulation and measurement experimental results showing very close (overlapping) distributions. Each graph shows the ECDFs of the latency of the measurement (solid line) and the simulation experiments (dashed lines).

Even though Figure 5 provides us with an excellent visual hint of the simulation model validity, as latency ECDFs are overlapping in all cases, we would like to have statistical confidence on how close those distributions are. To accomplish this statistical comparison we used the two-sample Kolmogorov-Smirnov (KS) test [28]. This non-parametric Goodness-of-Fit test [29], checks whether both samples come from populations with identical distributions. It tests for any violation of that null hypothesis – different medians, different variances, or different distributions.

The challenge to perform the KS test was that it is sensitive to large samples. As the number of requests processed during each experiment test is substantial, we applied an approach used in other studies to adapt the test to be used with large samples [30], [31]. We selected $1,000$ random samples of size 30 from each result dataset (i.e., from the simulation and measurement experiments), obtained the p-values for the KS test applied to each pair of samples and then calculated the average p-value from the $1,000$ p-values that we had. Table II presents those results.

TABLE II
RESULT OF KS TESTS COMPARING SIMULATION AND MEASUREMENT EXPERIMENTS REGARDING THE OBSERVED LATENCY DISTRIBUTIONS. VALUES WERE ROUNDED TO THE THREE MOST SIGNIFICANT DIGITS.

	1 Node	2 Nodes	4 Nodes
GCI Off	0.453	0.388	0.329
GCI On	0.439	0.394	0.295

As none of the p-values presented in Table II are small enough to reject the KS test's null hypothesis (e.g., 5%)[7], we conclude that the simulator and system experiment latency results come from populations with the same distribution. The populations do not differ in median, variability or the shape of the distribution.

V. RELATED WORK

Garbage Collection Coordination: Terei and Levy defined BLADE [32], which is an API that allows developers to leverage existing failure recovery mechanisms in distributed systems to coordinate garbage collection and bound latency. Taurus [33] is a mechanism to reduce tail latency by coordinating garbage collection in a distributed system. The later is a JVM replacement which can run unmodified Java applications and enforces cluster-wide user-defined coordination policies. Both BLADE and Taurus leave developers with the task of describing the coordination, which requires knowledge about the application, its workload, the environment, besides specific tuning. GCI is a more straightforward and easy-to-use mechanism which plugs in the application. It neither relies on cluster-wide coordination nor changes in the runtime.

Decrease Latency Variability: In [6], Dean, and Barroso described their efforts to build predictable latency tail latency in Google's interactive services. They describe techniques to deal with the problem, for instance, send the same requests to multiple servers (i.e., hedged requests), re-issuing slow requests to a different host or replicating data needed to answer a request. Related techniques have been used to improve

[7]From the original p-values we observed, on average, that 2% of them failed to reject the KS test null hypothesis

the performance of cloud data stores like Cassandra [34] and in data-parallel computation frameworks like MapReduce/Hadoop [35]. Our work is complementary to this body of work. Their approaches usually look at each node as a black-box to enhance the overall latency of distributed systems. GCI actuates within the node, transforming the non-deterministic impact caused by the garbage collection into a temporary node failure. This behavior allows GCI to be used in situations where requests cannot be duplicated and sent to multiple servers. Nevertheless, GCI could be used in conjunction with the techniques explored in others' work.

Improve Garbage Collection: There has been a large amount of work on improving pause times of garbage collectors [36], [37]. Proposed improvements are focused on algorithms [25], [38], application [37] or hardware classes [39]. Our work is orthogonal to this body of work, as GCI is not an approach to garbage collection, but a new approach to dealing with its performance impact on cloud services. GCI will rely on whatever garbage collector is running on the runtime. Faster collections will lead to smaller periods of node unavailability.

Runtime Tunning: Much work has been done to improve the performance of JVM applications by exploring the JVM configuration space either manually [40] or automatically [41]. Results showed that correctly tuning the system can increase JVM performance regardless of the architecture of the system. GCI is an approach to deal with GC interventions, and those are going to happen, even if the service had been tuned perfectly. That said, if the target service can be tuned and improve its overall performance, GCI would only make its tail latency better.

VI. Conclusions and Future Work

In this paper we propose and evaluate the Garbage Collector Control Interceptor (GCI) – a practical approach to improve the tail latency of cloud services by avoiding latency spikes caused by non-deterministic garbage collector activity. GCI controls garbage collector interventions and use available load shedding mechanisms to avoid processing the request during such interventions. In practice, cloud load balancers would transparently route shed requests to available service instances.

GCI is entirely agnostic regarding the target service code and load. Its usage neither requires manual configuration nor understanding of the runtime system's internals. Furthermore, GCI executes in a completely decentralized manner and relies on only two RTE-specific API calls, making it easy to be implemented.

We implemented GCI for the Java programming language and evaluated its effectiveness using a stateful service prototype. To assess GCI's suitability to cloud scale, we simulated small (4 nodes) and large (64 nodes) deployments. Our results showed that using GCI reduces the impact of the garbage collection on the service latency for both deployments with no throughput loss.

We are aware that the stateless services are an essential part of the cloud. We understand that many latency-sensitive microservices and serverless deployments are stateless and also suffer from the garbage collection impact. Extend our evaluation and improve GCI to deal with this case is our main future work.

Furthermore, we would like to perform a broader evaluation of our approach. That includes measuring GCI impact on real-world stateful services, such as Elasticsearch and Cassandra, as well as comparing GCI with other techniques to decrease latency variability (i.e., Elasticsearch's default GC tuning and Cassandra's dynamic snitch [42]). Finally, regarding the load, we plan to use the industry-standard Yahoo! Cloud Services Benchmark [43].

Acknowledgment

This work was supported by the Instituto Federal de Alagoas (IFAL) and CAPES– Brazilian Federal Agency for Support and Evaluation of Graduate Education. This work has also been partially supported by the project ATMOSPHERE (atmosphere-eubrazil.eu), funded by the Brazilian Ministry of Science, Technology and Innovation (Project 51119 - MCTI/RNP 4th Coordinated Call) and by the European Commission under the Cooperation Programme, Horizon 2020 grant agreement no 777154. Furthermore, we would like to thank the ePol/PF project, for the sponsorship.

References

[1] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 1–18. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509515

[2] C. Humble, "Twitter Shifting More Code to JVM, Citing Performance and Encapsulation As Primary Drivers," https://www.infoq.com/articles/twitter-java-use, 2011, online; Accessed: 2017-05-07.

[3] J. Verlaguet and A. Menghrajani, "Hack: a new programming language for HHVM," https://code.facebook.com/posts/264544830379293/hack-a-new-programming-language-for-hhvm/, 2014, online; Accessed: 2017-05-07.

[4] S. T. Krishnan and J. U. Gonzalez, *Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects*, 1st ed. Berkely, CA, USA: Apress, 2015.

[5] H. Li, *Introducing Windows Azure*. Berkely, CA, USA: Apress, 2009.

[6] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408794

[7] F. Xian, W. Srisa-an, and H. Jiang, "Garbage collection: Java application servers' achilles heel," *Sci. Comput. Program.*, vol. 70, no. 2-3, pp. 89–110, Feb. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2007.07.008

[8] A. W. v. d. Vaart, *Asymptotic Statistics*, ser. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1998.

[9] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, "The Java Virtual Machine Specification, Java SE 10 Edition," https://docs.oracle.com/javase/specs/jvms/se10/html/index.html, 2018, online; Accessed: 2018-06-27.

[10] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed, "Scuba: Diving into data at facebook," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1057–1067, Aug. 2013. [Online]. Available: http://dx.doi.org/10.14778/2536222.2536231

[11] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan, "Speeding up distributed request-response workflows," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 219–230, Aug. 2013. [Online]. Available: http://doi.acm.org/10.1145/2534169.2486028

[12] J. Dean, "Achieving rapid response times in large online services," 2012. [Online]. Available: https://research.google.com/people/jeff/latency.html

[13] G. Phipps, "Comparing observed bug and productivity rates for java and c++," *Softw. Pract. Exper.*, vol. 29, no. 4, pp. 345–358, Apr. 1999. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1097-024X(19990410)29:4⟨345::AID-SPE238⟩3.0.CO;2-C

[14] D. Fireman, R. Lopes, and J. A. Brunet Monteiro, "Using load shedding to fight tail-latency on runtime-based services," in *Brazilian Symposium on Computer Networks and Distributed Systems*, 2017.

[15] R. Fielding and J. Reschke, "RFC 7231 - HTTP/1.1 Semantics and Content," https://tools.ietf.org/html/rfc7231, 2014, online; Accessed: 2017-05-14.

[16] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[17] C. Gormley and Z. Tong, *Elasticsearch: The Definitive Guide*. O'Reilly Media, Inc., 2015.

[18] S. Madappa, V. Nguyen, S. Mansfield, S. Enugula, A. Enugula, and F. Siddiqi, "Caching for a global netflix," March 2016, [Online; posted 01-March-2016]. [Online]. Available: https://medium.com/netflix-techblog/caching-for-a-global-netflix-7bcc457012f1

[19] H. Salhi, F. Odeh, R. Nasser, and A. Taweel, "Open source in-memory data grid systems: Benchmarking hazelcast and infinispan," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17. New York, NY, USA: ACM, 2017, pp. 163–164. [Online]. Available: http://doi.acm.org/10.1145/3030207.3053671

[20] R. Daigneau, *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*, 1st ed. Addison-Wesley Professional, 2011.

[21] Java™ Platform Standard Ed. 8, "MemoryPoolMXBean," https://docs.oracle.com/javase/7/docs/api/java/lang/management/MemoryPoolMXBean.html, 2014, online; Accessed: 2017-05-12.

[22] M. Gorelick and I. Ozsvald, *High Performance Python: Practical Performant Programming for Humans*. O'Reilly Media, 2014. [Online]. Available: https://books.google.com.br/books?id=bIZaBAAAQBAJ

[23] P. Shaughnessy, *Ruby Under a Microscope: An Illustrated Guide to Ruby Internals*. San Francisco, CA, USA: No Starch Press, 2013.

[24] W. Reese, "Nginx: The high-performance web server and reverse proxy," *Linux J.*, vol. 2008, no. 173, Sep. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1412202.1412204

[25] D. Detlefs, C. Flood, S. Heller, and T. Printezis, "Garbage-first garbage collection," in *Proceedings of the 4th International Symposium on Memory Management*, ser. ISMM '04. New York, NY, USA: ACM, 2004, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/1029873.1029879

[26] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "Wake up and smell the coffee: Evaluation methodology for the 21st century," *Commun. ACM*, vol. 51, no. 8, pp. 83–89, Aug. 2008. [Online]. Available: http://doi.acm.org/10.1145/1378704.1378723

[27] R. Jain, *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.*, ser. Wiley professional computing. Wiley, 1991.

[28] F. J. Massey, "The Kolmogorov-Smirnov test for goodness of fit," *Journal of the American Statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[29] D. G. Feitelson, *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press, 2015. [Online]. Available: http://www.cambridge.org/de/academic/subjects/computer-science/computer-hardware-architecture-and-distributed-computing/workload-modeling-computer-systems-performance-evaluation

[30] M. Carvalho and F. Brasileiro, "A user-based model of grid computing workloads," in *2012 ACM/IEEE 13th International Conference on Grid Computing*, Sept 2012, pp. 40–48.

[31] B. Javadi, D. Kondo, J. M. Vincent, and D. P. Anderson, "Discovering statistical models of availability in large distributed systems: An empirical study of seti@home," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 11, pp. 1896–1903, Nov 2011.

[32] D. Terei and A. A. Levy, "Blade: A data center garbage collector," *CoRR*, vol. abs/1504.02578, 2015. [Online]. Available: http://arxiv.org/abs/1504.02578

[33] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz, "Taurus: A holistic language runtime system for coordinating distributed managed-language applications," *SIGPLAN Not.*, vol. 51, no. 4, pp. 457–471, Mar. 2016. [Online]. Available: http://doi.acm.org/10.1145/2954679.2872386

[34] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015, pp. 513–527. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/suresh

[35] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855744

[36] C. Click, G. Tene, and M. Wolf, "The pauseless gc algorithm," in *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, ser. VEE '05. New York, NY, USA: ACM, 2005, pp. 46–56. [Online]. Available: http://doi.acm.org/10.1145/1064979.1064988

[37] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard, "Broom: Sweeping out garbage collection from big data systems," in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/hotos15/workshop-program/presentation/gog

[38] Azul Systems, "Java without the jitter: Achieving ultra-low latency," 2015. [Online]. Available: https://www.azul.com/files/Java_wo_Jitter_v6.pdf

[39] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen, "NumaGiC: a Garbage Collector for Big Data on Big NUMA Machines," in *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM SIGOPS, ACM SIGPLAN, ACM SIGARCH. Istanbul, Turkey: ACM, Mar. 2015, pp. 661–673. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01178790

[40] L. G. Silva, C. A. P. S. Martins, and L. F. W. Goes, "Jvm configuration parameters space exploration for performance evaluation of parallel applications," *IEEE Latin America Transactions*, vol. 13, no. 8, pp. 2776–2784, Aug 2015.

[41] S. Jayasena, M. Fernando, T. Rusira, C. Perera, and C. Philips, "Auto-tuning the java virtual machine," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, May 2015, pp. 1261–1270.

[42] J. Hewitt, *Cassandra: The Definitive Guide, 2nd Edition*. O'Reilly Media, Incorporated, 2016. [Online]. Available: https://books.google.com.br/books?id=yTc_nQAACAAJ

[43] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152