Using Load Shedding to Fight Tail-Latency on Runtime-Based Services

Daniel Fireman, Raquel Lopes, João Brunet

[1]Departamento de Sistemas e Computação
Universidade Federal de Campina Grande (UFCG)
Caixa Postal 10.106 – 58.109-970 – Campina Grande – PB – Brasil

danielfireman@lsd.ufcg.edu.br, {raquel, joao.arthur}@computacao.ufcg.edu.br

Abstract. *HTTP services written in managed runtime languages such as Java are popular nowadays. By relying on a runtime environment (RE), these services can benefit from safer code, cross-platform, etc. However, it is well known that RE's pauses due to garbage collection increase response times (i.e. tail latency). As modern services often rely on many remote calls, the overall performance turns out to be determined by the tail, instead of the average latency. To address this problem we propose an easy-to-use combination of load shedding and control of garbage collector interventions. We implemented and evaluated a prototype in Java. Our results show a reduction of tail latency by approximately* 40%*, while throughput and CPU utilization were negligibly impacted.*

Resumo. *Serviços HTTP escritos em linguagens baseadas em runtime são muito populares nos dias de hoje. Por se basearem em uma runtime, estes serviços podem usufruir de benefícios como código mais seguro, multiplataforma e etc. Todavia, é bem conhecido que pausas da runtime devido à coleta de lixo aumentam o tempo de resposta. Como serviços modernos frequentemente se baseiam em diversas chamadas remotas, o desempenho total do sistema acaba sendo determinado pela cauda da distribuição de latência, ao invés da sua média. Como solução para esse problema, propomos uma combinação de prevenção de carga e controle das intervenções causadas por coletores de lixo. Nós implementamos um protótipo em Java e avaliamos o seu desempenho diante de workloads sintéticas. Os resultados mostram uma redução de aproximadamente* 40% *na cauda de latência, mantendo praticamente a mesma vazão e utilização de CPU.*

1. Introduction

Imagine a client making a request on a single web service. Ninety-nine times out of a hundred that request will be returned within an acceptable period of time. But one time out of hundred it may not. If you look at the service's latency distribution, there is one large entry at the tail end. When the service does not send upstream requests, all it means is that one client gets a slower response occasionally.

Now, instead of one, let us imagine that a request will require response from 100 services. That changes everything about your system's responsiveness. Suddenly the majority of queries (i.e., $1 - 0.99^{100} = 63\%$) will take greater than 1 second. Hence,

temporary high-latency episodes may come to dominate overall service performance at large scale [Dean and Barroso 2013].

It is important to point out that, not only the temporary high-latency is a problem, but also the high variability on the tail of the distribution because it increases the unpredictability of the system latency. To better explain this problem, let us analyze Figure 1. It shows a concrete example of the service time's percentiles for a synthetic workload. For example, looking at the 99.9 percentile line, a point in the graph means that, at a given time (horizontal axis), 99.9% of the requests are below the corresponding latency (vertical axis). As expected, the more we increase the percentiles, the slower are the service times.

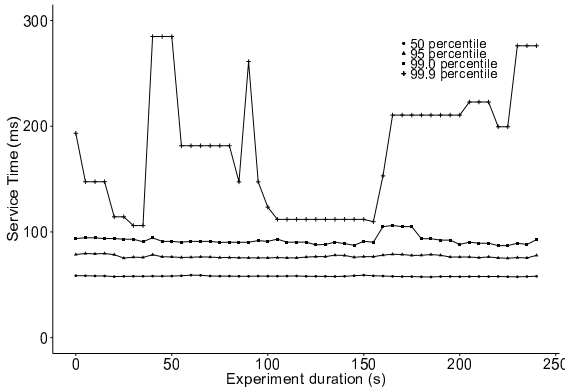


Figure 1. Service time of a single experiment run.

For clarity purposes, we would like to point out two main aspects of Figure 1. First, as you can see, the 99.9 percentile latency is substantially worse in comparison to the other percentiles. Also, the variability of the 99.9 percentile is substantially greater than the others. In this paper, we propose a solution to narrow down the 99.9 percentile line (reduce its variability) and bring it closer to the other percentiles (reduce service time tail) without significantly compromising the throughput and CPU utilization of the service.

To achieve this, we first had to understand the state-of-the practice on web services and what causes the aforementioned problem. In this context, there has been an increasing push for low latency and variability at the tail [Dean and Barroso 2013, Rumble et al. 2011]. To get a real feeling about these large scale services, a single Facebook page load can involve fetching hundreds of results from their distributed caching layer [Nishtala et al. 2013], while a Bing search consists of 15 stages and involves thousands of servers in some of them [Jalaparti et al. 2013]. These applications require latency in microseconds with tight tail guarantees.

Also, a large portion of workloads that are running in cloud datacenters are written in programming languages supported by managed runtime systems[Meyerovich and Rabkin 2013]. Companies such as Twitter [Humble 2011] and Facebook [Verlaguet and Menghrajani 2014] are writing most of their code in Scala and PHP. At the same time, cloud platforms such as Google AppEngine [Krishnan and Gonzalez 2015] and Microsoft Azure [Li 2009] are supporting managed languages as explicit targets. Finally, many web startups write their code in languages

such as Python, Ruby or JavaScript (which are all managed languages), as it allows them to iterate quickly. All these facts confirm that managed runtimes are popular and it does not seem a temporary situation.

Managed languages comes with a price though. Unfortunately, garbage collection (GC) is one of the main causes of high tail-latency [Dean and Barroso 2013]. That happens mainly due to two reasons: i) stop-of-the-world pauses [Gidra et al. 2013], in which applications are completely stopped during collection, and ii) CPU competition, as GC threads could run concurrently with the application.

In summary, managed languages are popular but the GC causes high tail-latency, as the ones observed in Figure 1. In order to eliminate garbage collection related latency with minimum impact on throughput, we propose an easy-to-use HTTP request interceptor to be added to web services in managed languages. Inspired by [Terei and Levy 2015], Garbage Collector Control Interceptor (GCI) controls garbage collection interventions and uses built-in HTTP load shedding mechanisms to deal with the requests that arrives during these interventions. We claim that the proposed solution has the following characteristics:

- Is application code and load agnostic;
- Requires little or no configuration;
- Requires no understanding of the runtime system's internals;
- Relies on vanilla HTTP for load shedding (RFC7231);
- Is completely open source.

We implemented a Java prototype of GCI to reduce garbage collection related latency in a stateless HTTP service. We compare the service performance with GCI on and off (baseline), both cases using the OpenJDK's default garbage collector scheme. Our results show that activating GCI leads to a 40% reduction in 99.9 percentile latency and a 5 times reduction in 99.9 percentile latency variability. All those benefits coming with a 1.2% decrease in throughput and 0.5% decrease CPU utilization in the worst case.

The remaining of this paper is organized as follows. Section 2 describes some key concepts. In Section 3 we describe the design and implementation of GCI. In Section 4 we describe our research questions and detail the performed evaluation and results. In Section 5 we describe related work and finally, we conclude in Section 6.

2. Background

2.1. Datacenter workload expectations

Nowadays, the workload running inside a datacenter is very heterogeneous, spanning from batch jobs that need no quality of service (QoS) guarantees to distributed HTTP services which are typically user-facing applications. These services are formed by many interacting web services that are lightweight, maintainable, and scalable [Rodriguez 2008]. Since these jobs are user-facing, they need QoS guarantees, especially regarding their latency. A single user request may travel among various service points before returning to the user again, touching dozens or even hundreds of servers. To allow these interactions between services without impacting the latency experienced by users, it is of utmost importance to consider not only the average latency of these services, but also their latency at the tail. In this environment, it is important to guarantee that even the 99.9 percentiles of the latency of the services are limited by rigid bounds [Terei and Levy 2015].

2.2. Opportunities of Managed Languages

Many of these web services are built using managed languages, i.e., languages whose programs execute on top of a runtime environment. Current examples of such languages are Java, Python and Ruby. The main reason of the popularity of these managed languages is probably time to market, since they often offer to developers some facilities such as dynamic typing, class resolution at runtime, reflection and garbage collection. Further advantages of managed languages include opportunities for dynamic optimization, especially when we consider long running services[Dean and Barroso 2013]. Jobs that run for a long time amortize the startup time delays impressed by the just-in-time compilation. Many managed runtime systems also compact memory during execution, which is important for long-running applications. This compaction avoids performance degradation from fragmentation and loss of locality.

2.3. State of Garbage Collection

Garbage collection is the automatic memory management mechanism of an executing program. It reduces the engineering overhead from explicitly dealing with pointers and eliminates many sources of errors. Even though runtime environment implementers have been working on building faster garbage collectors [Tene et al. 2011, Ugawa et al. 2014], pause times at the tail are still too long[Blackburn et al. 2008]. This is an important issue for those managed languages, since the impact of the garbage collection is often unpredictable and hard to debug. For instance, GC performance can vary greatly from system to system, or even over the lifetime of a single system [Soman et al. 2004].

As the GC impact is an unavoidable fact for those applications running on top of a runtime environment, it is important to have external tools to deal with it. Many languages provide support for programmable interfaces to interact with the garbage collector. For instance, Java Virtual Machine (JVM) supports the $System.gc()$ function, which suggests to the runtime to start the GC. It also has the Garbage Collection Notifications API extension, which supports notifying the application after a collection has completed[Oracle 2015]. JVM has no support to automatically disabling garbage collection. Microsoft .NET also has a Garbage Collection Notifications API, which offers a broad set of notification options [Microsoft 2015]. It also supports forcing a collection through $GC.Collect()$ function and disabling GC. We are aware of similar APIs in many other languages, for instance Python, Ruby and Go. Without these APIs the GCI would not be possible.

3. Garbage Collection Control Interceptor (GCI)

Changing the application code to deal with runtime pauses is difficult, as the GC behavior is often unpredictable and its impact is hard to debug. Furthermore, tuning the GC of a production system is hard. On the one side, it depends on characteristics of the load the service is subjected to, which can be very dynamic in worldwide distributed systems. On the other side, it depends on the service code, which can be deployed many times a day through continuous delivery pipelines.

To help on that matter we propose the Garbage Collection Control Interceptor (GCI). GCI is an HTTP request interceptor which controls garbage collector interventions and take action based on those events. For the purposes of this work, we define an

HTTP request interceptor as a piece of code that is executed before every single request received by an HTTP service. The concept itself is widely supported across most HTTP server frameworks, but its name can vary. Frameworks like Ruby on Rails, Java's J2EE and Spring call them Filters, whereas it is called Middleware amongst Go developers. Typically, interceptors can be activated with very minimum code changes (usually one line), offering a non-intrusive way of performing common processing desired for every HTTP request.

It is important to note that GCI is not an approach to garbage collection, but a new approach to dealing with its performance impact on HTTP services. GCI is a technique that is agnostic with regards to the HTTP service code and its load. It requires no (or very little) specific configuration or understanding of the runtime system's internals. As a consequence, it is easy to use. GCI relies on vanilla HTTP specification for load shedding (RFC7231) and is completely open source[1].

Part of the GCI request processing is to decide when GC must run. Our proposal is quite simple: monitor the utilization of the runtime's heap pool(s) and when it reaches a certain threshold, it is time to collect the garbage. With this in mind, Algorithm 1 shows the pseudocode of the interceptor.

```
Input: Resp: HTTP response which is going to be sent to the client
Output: Whether to continue the request handling process
1  begin
2      if not ShouldAcceptRequest() then
3          Resp.SetStatusCode(503)
4          Resp.SetHeader("Retry − After", EstimateUnavailability())
5          return False
6      end
7      if SampleHeapUsage() then
8          if GetHeapUsage() > SHEDDING_THRESHOLD then
9              StopAcceptingRequests()
10             concurrent
11                 WaitOutstadingRequests()
12                 GC()
13                 StartAcceptingRequests()
14             end
15         end
16     end
17     return True
18 end
```

Algorithm 1: Garbage Collector Control Interceptor Pseudocode

In summary, the GCI controls when GC must run, guaranteeing that no request competes with GC for CPU or is delayed by having to wait in queues during the GC intervention. The algorithm is simple and for practical purposes it can be implemented to work with Java, Go, Ruby, and Python runtimes (but not restricted to them).

The processing of every request starts by checking whether the incoming request

[1] Available at https://github.com/danielfireman/gci

should be processed. If the request is allowed to be processed, the next step is to verify whether the usage of memory pools reached the specified threshold. If so, the system stops receiving new requests and a concurrent code block starts (so the request being executed is not blocked). This concurrent block waits for all the outstanding requests to be processed to then and forces the activation of the garbage collector. When the collection finishes, the system starts accepting requests again. We are going to dive into algorithm details in the next subsections.

3.1. Shedding requests

When a request must not be accepted ($ShouldAcceptRequest()$ returns false), the response is modified and the interceptor method returns false. This returned value determines the end of the request processing and immediate delivery of the HTTP response. The modified response has a 503 status code (Service Unavailable). As per RFC7231, the service unavailable status indicates that the service is currently unable to handle the request due to a temporary overload or scheduled maintenance, which will likely be alleviated after some delay [Fielding and Reschke 2014].

Furthermore, GCI shed responses always have the Retry-After header set. The service unavailability duration can be estimated using linear extrapolation from previous events. This approach uses the values of the previous utilization of target memory pools as predictors (or the overall heap, for non-generational GC schemes).

Shed requests may be immediately resent to another server. In practice, this is done transparently by client APIs or load balancers. Also, the Retry-After response header field suggests an appropriate amount of time for the client to wait before sending requests again to the unavailable service.

3.2. Sampling Memory Usage Check

All languages considered export methods to fetch information about memory utilization. Even though checking memory usage does not incur in a prohibitive cost when done once in a while, check memory usage at each request would make GCI unusable, especially in high load production deployments as ours. To decrease this overhead, GCI uses a sampling window. Thus, the GCI knows the memory utilization information related to the most recent window.

The $SampleHeapUsage()$ method checks the sampling window and returns true whether it is time to perform a memory utilization check, i.e, if a new window must be considered. The size of this window varies based on the previous number of requests processed between consecutive garbage collections. Using a number of requests is better than a time interval for two reasons. Firstly, it is less impacted by load peaks and, secondly, it does not trigger unnecessary checks due to load valleys. All these values related to the memory utilization monitoring are configurable.

3.3. Controlling Garbage Collector Activity

When the shedding threshold has been reached, no more requests can be accepted until garbage collection operation finishes. This is done by calling the $StopAcceptingRequests$, which makes $ShouldAcceptRequest$ return false until $StartAcceptingRequests$ call.

Garbage collector activity might incur in CPU competition or stop-of-the-world pauses. For this reason, it is important to ensure that garbage collection does not occur while processing requests. Thus, GCI must wait for all outstanding requests to finish before collecting the memory garbage. Furthermore, these actions must not block the request being processed (which would incur in latency increase). This is the reason for the $concurrent$ block (lines 10-14), which could, for example, be implemented in Java by running the code in a new thread.

GCI's garbage collection control depends on: i) triggering (forcing) a garbage collection and ii) avoiding automatic garbage collection. The former is expressed in the pseudocode as $GC()$. As an example, the Java implementation of the $GC()$ method can be executed through a $System.gc()$ call. All target languages expose similar functions to trigger garbage collections.

Disabling automatic garbage collections can be a problem for languages like Java, which lack of programmatic ways of doing so. One way to deal with this limitation is to make sure that automatic GC interventions will be as infrequent as possible. How to do that? Since automatic GC occurs when the memory fills up, all we need to do is to configure the runtime with the maximum possible heap and/or memory pools (for generational GCs). Fortunately, other languages like Python, Ruby and Go have easy ways to entirely disable automatic garbage collection, thus not requiring this workaround.

It is important to notice that GCI relies on the garbage collection scheme available. It is orthogonal to any GC specifics as well as to any custom configuration or tuning performed.

4. Evaluation

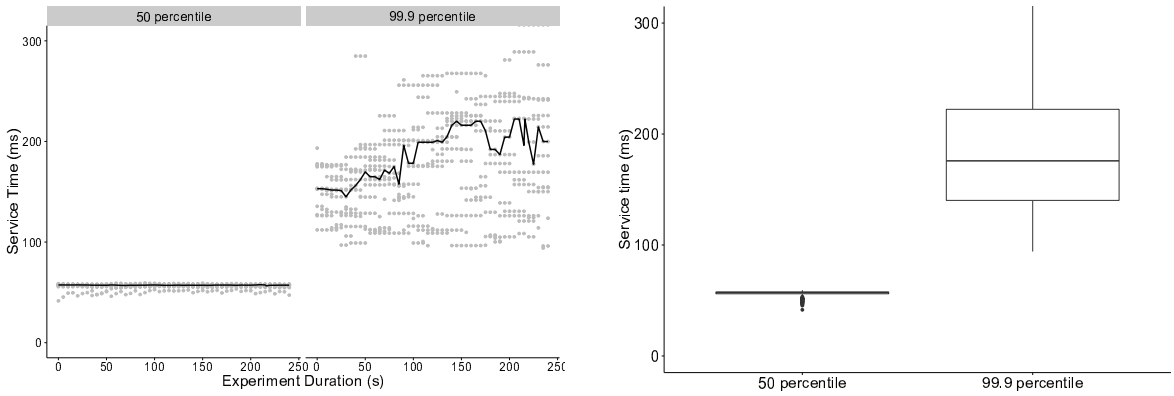
4.1. Experimental Setup

Let us start the evaluation by remembering the two main problems GCI aims to solve. First, Figure 2(a) illustrates service times for a synthetic workload over time. The black line is the average service times (median). The difference between the 99.9 and 50 percentile ranges from $72.69ms$ to $200.20ms$, which represents a meaningful quality of service degradation.

Second, in Figure 2(b), we present box plots that summarize the variability of both 50 and 99.9 percentile cases. The interquartile range (IQR) of the 99.9 percentile is $70.4ms$, which is in contrast with 50 percentile IQR (close to zero). That makes service time's predictability very difficult for the tail end, affecting capacity planning and SLOs definition, for example.

We conducted a 1-factor design experiment to evaluate Garbage Collector Interceptor impact on the 99.9 percentile service time and its variability. The activation of GCI was the only factor (GCI On and Off) and the service time was the dependent variable. We also chose a simple service request handling flow: a CPU intensive operation and small pause, simulating a blocking I/O call. This flow would represent, for instance, a query to a database and processing results before sending the response back to the client.

We replicated each experiment 15 times. Each one of them last around 8 minutes. We discarded the first 4 minutes of each experiment to minimize JVM warm-up



(a) 50 and 99.9 service time percentiles (b) Variability summary

Figure 2. Average service time

effects [Blackburn et al. 2008]. A client running on a different machine generated a constant load of 70 requests per second. Service times were measured for later evaluation.

The HTTP server executed in a virtual machine with 2 VCPUs (2660 MHZ) and 4GB of RAM running on a Ubuntu Linux (kernel version 4.4.0-53-generic). The server was executed using the OpenJDK 1.8.0_11 64-Bit Server VM (build 25.111-b14). Furthermore, to activate GCI in Java we needed to avoid as much as possible the automatic garbage collection. Because of that, we fixed heap size to 1 GB (i.e., setting -Xms1024m and -Xmx1024m) and split the heap equally between young and old generations pools (i.e. -XX:NewRatio=1).

4.2. Service Time Improvements

As we expect GCI to narrow and decrease the service time distribution tail, we drove our evaluation based on the following null hypotheses:

$H_{0,1}$: GCI does not improve the 99.9 percentile of the service time.

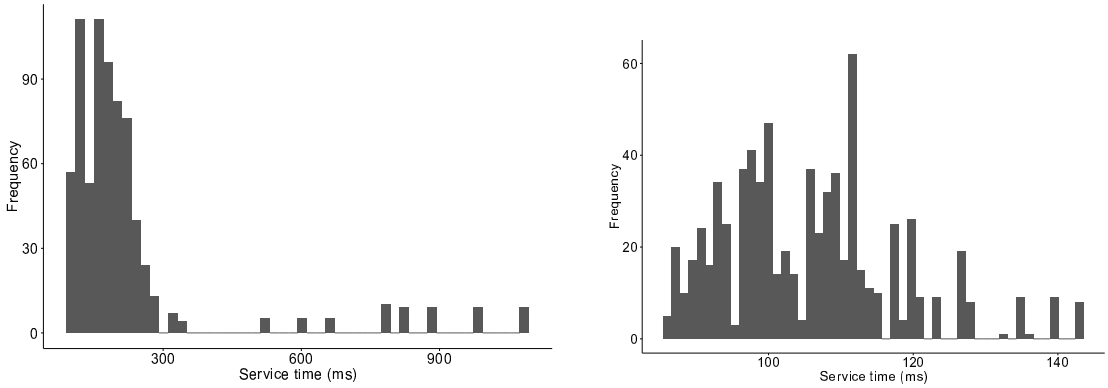
$H_{0,2}$: GCI does not reduce the 99.9 percentile variability of the service time.

We measured the 99.9 percentile service time with both GCI off and on (Figures 3(a) and 3(b)) to address these hypotheses.

Histograms in Figure 3 give the shape of the 99.9 percentile distributions of the service times. Please, be aware that axes are different. As you can note, the GCI reduces the tail of the 99.9 service time percentile, as the distribution becomes more symmetric. Furthermore, the GCI decreases the service time variation, once the range is narrower (from $]0, 1200]ms$ to $]0, 150]ms$).

In summary, we could say that GCI leads to the following benefits:

- Faster service times: within the 99.9 service time percentile, the median decreased from $175.8ms$ to $105.2ms$ and
- Predictable service times: within the 99.9 service time percentile, the IQR went down from $82.1ms$ to $14.7ms$. With less variation it easier to predict even the maximum expected service times.



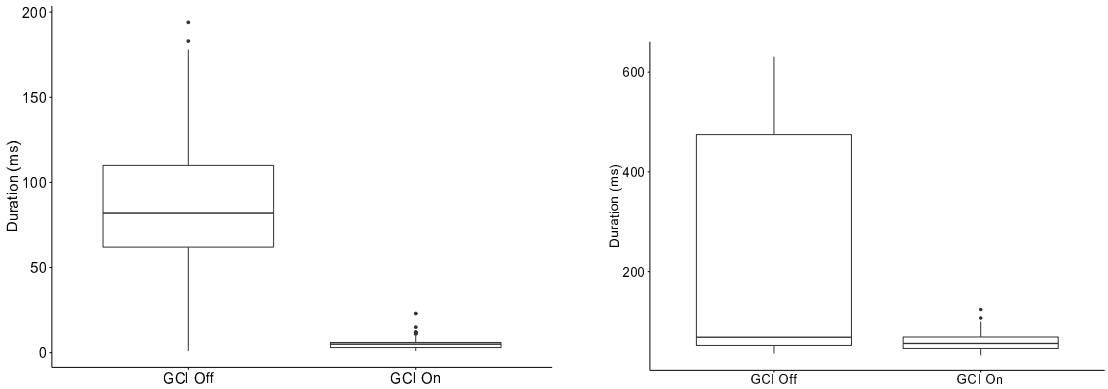
(a) GCI Off (b) GCI On

Figure 3. Histogram of 99.9 service time percentile

To statistically confirm these results, we carried out one-sided version of the non-parametric Mann-Whitney U Test [Hettmansperger 2011]. We chose this test because both samples do not come from a normal distribution, as confirmed by the very low p-values of Shapiro-Wilk Test (8.37^{-16} and 1.005^{-14}) [Shapiro and Wilk 1965]. Based on the result of the Mann-Whitney U test (p-value $< 2.2^{-16}$), we refute hypothesis $H_{0,1}$, with an estimated improvement around $71.42ms$. As for hypothesis $H_{0,2}$, we had already shown evidences of its refutal based on the I.Q.R. improvement.

4.3. Understanding the Impact on GC Behavior

As GCI controls garbage collections and sheds requests, it ends up changing GC's throughput and footprint [Sun Microsystems 2009]. To illustrate that, we present in Figure 4 an aggregated summary of GC interventions considering all experiment runs with GCI on and off.



(a) Minor GCs (b) Major GCs

Figure 4. Aggregated garbage collection activity

We analyzed the time span of the two garbage collection types: major and minor. The Java runtime's heap is managed in generations (young and tenure), which are memory pools holding objects of different ages. When a generation is full, the JVM triggers

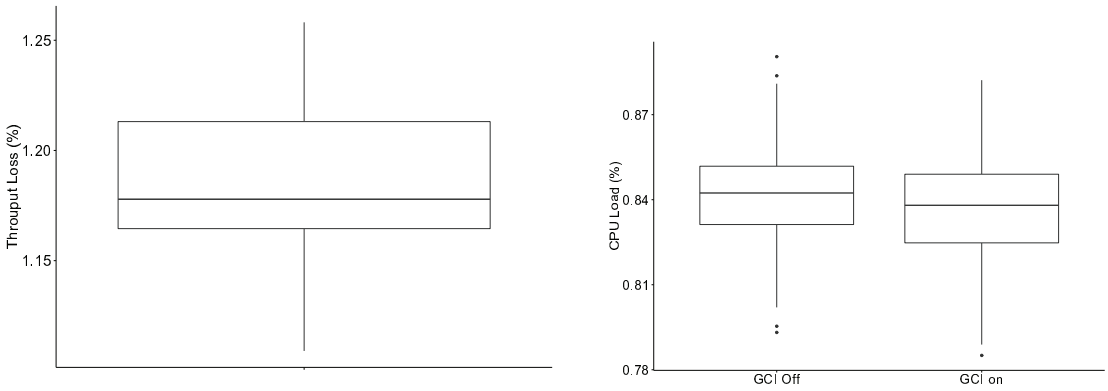
garbage collection. For the young generation, it causes a minor garbage collection. During this process, the JVM moves the surviving objects to the tenured generation. When, eventually, the tenured generation is full, the JVM triggers a major collection. Major garbage collections usually last much longer than the minor ones because a significantly larger number of objects are involved.

Minor garbage collections are represented in Figure 4(a) and it is easy to confirm that activating GCI leads to shorter garbage collections and also less variability.

Activating GCI also leads to shorter major collections and also less variability (Figure 4(b)). This happens because our solution triggers minor and major garbage collections when either pool reaches the shed threshold of utilization. By forcing garbage collection before the the automatic increasing of the generations size, GCI prevents long runs of the major garbage collection.

4.4. Overhead

We are aware that our approach deals with a trade-off: by shedding some requests to improve service time we decrease the overall service throughput. Naturally, our solution could not significantly compromise throughput or CPU load. For this reason, we also investigated the overhead that GCI generates. We calculated throughput loss as the ratio between shed requests and total requests (%).



(a) Throughput loss when GCI is on

(b) CPU Load

Figure 5. Throughput and CPU load aggregated summary

Figure 5(a) confirms that GCI does impact on throughput but not substantially. With 95% of statistical confidence, the mean throughput loss is $[1.16605\%, 1.209268\%]$. In a nutshell, GCI leads to less than 1.2% throughput loss on average. We consider this to be a good trade-off given the improvements in performance.

As GCI controls which requests to shed, one might wonder how it impacts on CPU utilization. Figure 5(b) presents data on this matter. As we can see, GCI leads to a small decrease in CPU utilization. To be more precise, with 95% of statistical confidence, the mean CPU load reduced from $[84.0\%, 84.26\%]$ to $[83.54\%, 83.79\%]$, which is a marginal impact.

4.5. Threats to Validity

This paper presents preliminary results of garbage collection interceptor. It is limited by some factors that can jeopardize external validity in the context of the experimental design. We are confident about our measurements: they were collected using proper instrumentation and we discarded warm up phase. However, we still must state the following threats to external validity:

- We are using one application. We do not know the extent to which the positive results of this study can be generalized to other applications;
- We focused on Java Virtual Machine. We are confident about the possibility of implementing our solution in other languages, but we cannot guarantee that the results of our study are extended to all such languages, especially when we consider the particularities of the garbage collection mechanisms implemented by other runtime environments;
- One machine configuration. For the sake of simplicity we considered only one type of server running the application. Different types of servers, with different memory sizes may lead to different results.

We hope to improve these threats in the future by considering other applications, languages and server configurations. It is important to mention that another threat would be that we fixed the workload to a constant load. In real environments, workloads are more dynamic and less predictable. However, we can consider that there is an auto scaling system adding and releasing resources from the application dynamically. That would guarantee that all the servers running are being highly and equally used, as the server we set up in our experiment.

5. Related Work

Some previous work also tackle the problem of high tail-latency due to the garbage collection done by the runtime environments. We are not aware, however, of a previous work that provides such simple and easy to use solution to deal with this problem by controlling garbage collection interventions and shedding the incoming requests during these interventions.

In [Terei and Levy 2015], Terei and Levy defined BLADE, which is an API that leverages existing failure recovery mechanisms in distributed systems to coordinate garbage collection and bound latency. They investigated two usage scenarios: an HTTP load-balancer and the Raft consensus algorithm. In both cases, latency at the tail using BLADE is up to three orders of magnitude better. In order to take advantage of BLADE, applications must be modified and use the BLADE API, which excludes all the legacy applications from bounding latency. When developing an application using BLADE API, developers need to know about memory management, garbage collection and other details. Furthermore, BLADE has only been implemented for Go language so far. Our solution pursues similar goals as BLADE, but by different means. By focusing on HTTP Services we can provide a much simpler and easy-to-use service, that plugs in the application, i.e. it is not part of the application by construction.

In [Maas et al. 2016] authors propose Taurus, which is a mechanism to reduce tail latency by coordinating garbage collection in a distributed system. Taurus is a JVM

replacement which can run unmodified Java applications and enforces user-defined coordination policies. They evaluated Taurus using two different applications: Spark [65] and Cassandra [33].

Both, BLADE and Taurus coordinate runtime activities considering the whole distributed application to avoid service disruptions. These systems leave developers with the task of programming or describing the coordination itself, which requires knowledge about the application, its workload, the environment, besides specific tuning. Another downside of Taurus is that it relies on a modified version of the JVM, which will need to be updated and maintained as any other component of the system. Our solution is orthogonal to these solutions, since it focus on coordinating each independent endpoint of the application independently by controlling GC executions and shedding requests during garbage collection interventions.

6. Conclusions and Future Work

This paper proposes Garbage Collection Control Interceptor (GCI) - a request interceptor which aims to control garbage collector interventions to improve its performance impact on HTTP services. GCI is a load and app-independent technique. It requires a marginal configuration effort and no understanding of the runtime system's internals. As a consequence, it is easy to use and port to other languages. GCI relies on vanilla HTTP specification for load shedding (RFC7231) and it is completely open source.

We evaluated GCI on a stateless HTTP Service with a constant workload. Experimental results showed that GCI significantly (40%) reduces 99.9 service time percentile and its variation (interquartile range decreased by 82%). That means that our work is a step towards faster and more predictable service times. Results also demonstrated a very small negative impact on throughput and CPU load.

Our main future work is to perform a broader evaluation of our approach. That includes measuring GCI impact on real world HTTP Services, such as Elasticsearch. It also includes to evaluate GCI implementations regarding other programming languages, such as Python, Ruby and Go. At last, even though GCI impact on throughput is marginal, we intend to act on the shed requests to reduce such impact to a minimal one.

Acknowledgments: This work was conducted during a scholarship supported by CAPES – Brazilian Federal Agency for Support and Evaluation of Graduate Education. This work is also sponsored by the agreement between UFCG and ePol/PF.

References

Blackburn, S. M., McKinley, K. S., Garner, R., Hoffmann, C., Khan, A. M., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanovik, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2008). Wake up and smell the coffee: Evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89.

Dean, J. and Barroso, L. A. (2013). The tail at scale. *Commun. ACM*, 56(2):74–80.

Fielding, R. and Reschke, J. (2014). Rfc 7231 - http/1.1 semantics and content.

Gidra, L., Thomas, G., Sopena, J., and Shapiro, M. (2013). A study of the scalability of stop-the-world garbage collectors on multicores. In *Proceedings of the Eighteenth*

International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pages 229–240, New York, NY, USA. ACM.

Hettmansperger, T. P. (2011). *Robust nonparametric statistical methods*. CRC Press.

Humble, C. (2011). Twitter Shifting More Code to JVM, Citing Performance and Encapsulation As Primary Drivers.

Jalaparti, V., Bodik, P., Kandula, S., Menache, I., Rybalkin, M., and Yan, C. (2013). Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 219–230, New York, NY, USA. ACM.

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

Li, H. (2009). *Introducing Windows Azure*. Apress, Berkely, CA, USA.

Maas, M., Asanović, K., Harris, T., and Kubiatowicz, J. (2016). Taurus: A holistic language runtime system for coordinating distributed managed-language applications. *SIGOPS Oper. Syst. Rev.*, 50(2):457–471.

Meyerovich, L. A. and Rabkin, A. S. (2013). Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 1–18, New York, NY, USA. ACM.

Microsoft (2015). .NET garbage collection notifications api.

Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. (2013). Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA. USENIX Association.

Oracle (2015). Java JMX garbage collection notification api.

Rodriguez, A. (2008). Restful web services: The basics. *Online article in IBM DeveloperWorks Technical Library*, 36.

Rumble, S. M., Ongaro, D., Stutsman, R., Rosenblum, M., and Ousterhout, J. K. (2011). It's time for low latency. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 11–11, Berkeley, CA, USA. USENIX Association.

Shapiro, S. S. and Wilk, M. B. (1965). An analysis of variance test for normality (complete samples). *Biometrika*, 3(52).

Soman, S., Krintz, C., and Bacon, D. F. (2004). Dynamic selection of application-specific garbage collectors. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 49–60, New York, NY, USA. ACM.

Sun Microsystems (2009). Java SE 6 HotSpot virtual machine garbage collection tuning.

Tene, G., Iyengar, B., and Wolf, M. (2011). C4: The continuously concurrent compacting collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 79–88, New York, NY, USA. ACM.

Terei, D. and Levy, A. A. (2015). Blade: A data center garbage collector. *CoRR*, abs/1504.02578.

Ugawa, T., Jones, R. E., and Ritson, C. G. (2014). Reference object processing in on-the-fly garbage collection. In *Proceedings of the 2014 International Symposium on Memory Management*, ISMM '14, pages 59–69, New York, NY, USA. ACM.

Verlaguet, J. and Menghrajani, A. (2014). Hack: a new programming language for HHVM.